

Università degli Studi di Pisa
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

UMLsdd: un editor per i diagrammi di sequenza UML

Candidato:

Davide Anastasia

Relatori:

Prof. Andrea Domenici

Prof.ssa Cinzia Bernardeschi

Anno Accademico 2003-2004

Ai miei genitori

Questo elaborato è stato impaginato utilizzando lo strumento professionale \LaTeX 2e. Tutti i grafici sono stati realizzati utilizzando XFig, tranne i diagrammi di sequenza, che sono stati realizzati con UMLsdd, il programma oggetto di questo elaborato. La programmazione di UMLsdd è stata facilitata dall'utilizzo dell'IDE Anjuta. Tutte le librerie utilizzate sono rilasciate sotto licenza LGPL. I miei sorgenti sono rilasciati sotto GPL. Tutto il lavoro è stato svolto su Linux utilizzando il compilatore GCC. L'ultima revisione di questo documento è del 6 dicembre 2004.

Indice

Introduzione	iii
1 UML	1
1.1 Storia dell'UML	2
1.2 Il linguaggio UML	5
1.3 Diagramma di sequenza	8
1.3.1 Semantica	8
1.3.2 Notazione	8
1.3.3 Opzioni di rappresentazione	9
1.3.4 Linea di vita degli oggetti	9
1.3.5 Attivazione	11
1.3.6 Messaggi e stimoli	11
1.3.7 Transizioni temporali	14
2 Librerie di supporto	15
2.1 GTKmm	15
2.1.1 Perché GTKmm?	15
2.1.2 Come creare le interfacce con le GTKmm	16
2.1.3 Un esempio di interfaccia	20
2.2 SigC++	24
2.2.1 Perché SigC++?	24
2.2.2 Come gestire i segnali	25

2.3	Libgnomecanvasmm	29
2.3.1	La gestione degli eventi	30
2.3.2	La definizione delle proprietà degli oggetti	32
2.4	STL - Standard Template Library	33
2.4.1	List	33
2.4.2	Iteratori	34
2.4.3	Algoritmi standard	35
2.4.4	Oggetti funzione	37
3	Scelte di progetto	41
3.1	Gli elementi di modello	41
3.1.1	Movimento e aggiornamento del grafico	44
3.1.2	Reshaping	49
3.2	Canvas Manager	50
3.2.1	Inserimento di un elemento di modello	53
3.2.2	Eliminazione di un elemento di modello	55
3.2.3	Cambio delle proprietà di un elemento di modello	58
3.3	Altre scelte	62
3.3.1	Considerazioni sul pattern Observer	62
3.3.2	Model-View-Controller (MVC)	63
3.4	UMLsdd	64
	Conclusioni e ringraziamenti	67
	Bibliografia	69

Introduzione

Questo elaborato descrive la realizzazione di una semplice applicazione per la creazione di *diagrammi di sequenza UML* e le scelte di progetto operate. Innanzi tutto vengono messe in evidenza le motivazioni che hanno portato alla scelta di determinate librerie, poi viene descritto il modo in cui sono state utilizzate tecniche avanzate di programmazione ad oggetti, come i *design patterns*, ed infine vengono giustificate alcune scelte di progetto nella scrittura e l'organizzazione del codice del programma.

Il lavoro è diviso in tre capitoli. Nel primo viene trattato l'UML: da un punto di vista qualitativo nella prima parte, accennando alla sua storia, lo scopo per cui è nato e i suoi utilizzi concreti [1]; nella seconda parte viene analizzata la sua struttura interna con particolare riferimento al diagramma di sequenza, di cui viene spiegata anche la notazione [2].

Nel secondo capitolo viene descritta la struttura e il funzionamento delle librerie utilizzate. Verranno mostrati esempi di codice alternati a descrizioni schematiche, evidenziando come alcune soluzioni possano essere collegate a pattern strutturali noti e consolidati [3]. I riferimenti più importanti in questo capitolo sono i siti internet e le documentazioni tecniche allegate ai sorgenti delle librerie, nonché alcuni tutorial presenti in rete, spesso scritti dagli stessi sviluppatori [4, 5, 6].

Il terzo capitolo descrive le problematiche affrontate nella scrittura del codice. Queste sono in maggior parte connesse con la gestione del canvas, poiché la libreria `libgnomecanvasmm` fornisce al programmatore pochi stru-

menti lasciandogli un altissimo grado di libertà. Verrà trattato in parti distinte l’inserimento e la cancellazione di un elemento del grafico e la gestione degli aggiornamenti in seguito a spostamenti o modifiche degli elementi, ponendo in risalto la gestione della coerenza visuale del grafico (cioè la capacità del grafico di non perdere mai la sua forma per prenderne una inconsistente). Nelle ultime pagine si parlerà del programma e del funzionamento a livello d’interfaccia utente, citando anche le caratteristiche del diagramma e ulteriori funzionalità desiderabili, che non sono state implementate.

Capitolo 1

UML

Lo *Unified Modeling Language* (UML) è un linguaggio grafico general-purpose di modellizzazione usato per specificare, costruire, visualizzare e documentare un sistema software[1].

L'utilizzo dell'UML può diventare fondamentale per la realizzazione di un buon progetto, il quale rappresenta un investimento sulla longevità del software che si va a realizzare, poiché rende semplici la manutenzione, il miglioramento e la correzione dei *bug*.

UML nasce per unificare l'esperienza passata con lo stato dell'arte per ottenere un'approccio standard alla modellizzazione del software, includendo concetti semantici, notazione e linee guida. Il suo uso è supportato da strumenti di modellizzazione visuali ed interattivi, che permettono anche la generazione automatica del codice¹. UML non specifica il tipo di processo di sviluppo, ma è inteso al supporto di tutte le tecniche di progettazione

¹Questa operazione è possibile, ma con alcune limitazioni, relative soprattutto alla scelta del modello di generazione. Ciò rappresenta un problema in particolar modo nei comportamenti dinamici, poiché diverse scelte del modello possono portare a generazioni automatiche completamente diverse. Il problema è meno sentito per le strutture statiche, che spesso risultano vincolate dal linguaggio di programmazione scelto.

object-oriented. Il linguaggio permette, infatti, di specificare sia strutture statiche che comportamenti dinamici: un sistema è modellizzato come una collezione di oggetti discreti che svolgono un compito per il beneficio di un utente esterno. Le strutture statiche descrivono la forma degli oggetti fondamentali del sistema e la loro implementazione, così come le relazioni tra gli oggetti. I comportamenti dinamici definiscono l'evoluzione temporale e le comunicazioni che avvengono allo scopo di conseguire l'obiettivo. Mostrare come il sistema è stato modellizzato da punti di vista differenti, ma correlati, permette di comprendere il progetto nella sua completezza. UML permette anche di dividere il progetto in *package*, cioè unità di lavoro separate all'interno dello stesso software. Le unità di lavoro concettuali del software possono poi diventare unità fisiche separate assegnate a diversi gruppi di una stessa squadra. Ovviamente in un contesto di questo tipo è fondamentale specificare le dipendenze e le relazioni tra i diversi package.

E' importante capire che *UML non è un linguaggio di programmazione*. Gli strumenti di progettazione possono generare il codice dal progetto UML in una moltitudine di linguaggi, così come alcuni strumenti permettono di generare grafici e modelli dal codice sorgente già esistente, anche se spesso questa operazione risulta assai complicata. UML non è nemmeno un linguaggio altamente formale: è un linguaggio di modellizzazione general-purpose. Per speciali domini di applicazioni è più appropriato utilizzare linguaggi specifici. UML è un linguaggio discreto e non è quindi adatto per i sistemi tempo-continui tipici dell'ingegneria e della fisica.

1.1 Storia dell'UML

UML nasce per semplificare e consolidare un largo numero di tecniche esistenti utilizzate per la progettazione di sistemi object-oriented.

Le prime tecniche di progettazione del software furono ideate negli anni '70 per poi raggiungere una larga diffusione negli anni '80. Era un periodo

caratterizzato dall'utilizzo di linguaggi classici come il Cobol ed il Fortran. Tra i metodi più diffusi c'era lo “*Structured Analysis and Structured Design*” ed alcune sue varianti. Le metodologie di questo periodo ebbero una significativa influenza nella progettazione di grandi sistemi, specialmente commissionati da enti aerospaziali o governativi. I risultati in molti casi non furono buoni come sperato, soprattutto per via di strumenti CASE molto poveri nelle funzionalità; ma in certi casi alcuni concetti si rivelarono di una certa utilità pratica. Per quanto riguarda il settore industriale, questo rimase sempre un po' restio ad adottare metodi di progettazione strutturati e strumenti CASE di supporto. Solitamente sviluppavano il software internamente in base ai propri bisogni, senza il tipico rapporto cliente/consumatore esistente nella caratterizzazione dei sistemi per gli enti governativi².

Il primo linguaggio di programmazione ad oggetti è stato *Simula-67*. Questo non ha mai riscosso un gran successo di per sé, ma ha avuto una grande influenza sulla progettazione di molti altri linguaggi di programmazione ad oggetti successivi.

Il movimento della programmazione ad oggetti esordì in seguito alla diffusione di *SmallTalk*, nei primi anni '80. Più tardi nacquero anche Objective C, C++, Eiffel e CLOS. Quelli maggiormente utilizzati attualmente sono Objective C e C++, insieme al linguaggio Java della Sun, nato successivamente. Cinque anni dopo l'ampia diffusione di Smalltalk, nell'arco di un breve periodo, furono pubblicati una serie di libri sulle metodologie di progettazione object-oriented³. Questa fase si concluse alla fine del 1990.

²Questo rapporto viene meno tra cliente finale e software house: il primo vuole un software che risponda a determinati criteri di semplicità e costo, mentre la casa produttrice sceglie i metodi che ritiene più efficaci per raggiungere questi obiettivi.

³Tra questi c'erano anche i libri “*Object-Oriented Analysis and Design with Application*” di Booch e “*Object-Oriented Modeling and Design*” di Rumbaugh. Questi diventeranno poi (insieme a Jacobson) i redattori della prima bozza di UML.

Solo 2 anni dopo fu pubblicato il libro di Jacobson, *the Objectory book*, che rappresentava una sintesi riorganizzata dei lavori precedenti. Tale libro utilizzava un approccio differente, focalizzato sui casi d'uso e sul processo di sviluppo. Nei cinque anni successivi fu pubblicata un'enorme quantità di libri sulle metodologie object-oriented, ognuno con un suo gruppo di concetti, definizioni, notazioni e terminologia. Alcuni di questi aggiungevano qualcosa di nuovo, ma per la maggior parte c'era grande somiglianza tra i vari autori. A questi nuovi libri si aggiungevano gli aggiornamenti di quelli principali.

In generale emerse un gruppo di concetti comuni insieme ad altri utilizzati da uno o due autori, ma non da tutti.

I tentativi iniziali di unificare i vari metodi ebbero spesso l'effetto di crearne di nuovi. La prima vera unificazione avvenne quando Rumbaugh incontrò Booch alla *Rational Software Corporation*, nel 1994. Incominciando a combinare i loro metodi, proposero una prima unificazione nel 1995. Nello stesso periodo anche Jacobson fu assunto alla Rational e incominciò a lavorare con Booch e Rumbaugh. Dall'unificazione dei loro lavori nacque lo *Unified Modeling Language* (UML). Questo fu reso possibile soprattutto dal particolare momento, che vedeva i tre maggiori autori di tecniche di progettazione object-oriented nella stessa azienda.

Nel 1996, lo *Object Management Group* (www.omg.org) emise una richiesta di proposte per approcci standard alla modellazione object-oriented. Gli autori di UML, con metodologisti e sviluppatori di altre compagnie, iniziarono a lavorare per elaborare una proposta che fosse attraente sia per i membri della OMG, sia per i tool makers, i metodologisti e gli sviluppatori che avrebbero poi dovuto diventarne utenti. Tutti gli sforzi confluirono nella versione finale di UML, presentata alla OMG nel settembre 1997.

Il prodotto finale fu il risultato della collaborazione di molte persone e fu accettato come standard dall'OMG nel novembre 1997. L'OMG si prese

quindi carico degli sviluppi successivi. Subito dopo l'adozione finale, furono pubblicati numerosi libri sull'argomento e molte aziende si dichiararono disponibili al supporto di questo standard.

1.2 Il linguaggio UML

UML è di supporto ad ogni fase della realizzazione del software, a partire dall'analisi dei requisiti fino alla programmazione vera e propria.

UML è composto da un certo numero di diagrammi, ognuno con la sua sintassi e semantica grafica. Quindi, in UML, un sistema viene descritto da punti di vista diversi. Ognuna di queste viste è un diagramma, composto dalle rappresentazioni grafiche degli *elementi di modello*. Diagramma per diagramma, la rappresentazione grafica dello stesso elemento di modello può variare (ad es. diversa è la rappresentazione di una classe in un diagramma delle classi ed in un diagramma di sequenza). Questa distinzione sarà chiara quando si vedranno alcuni esempi di diagrammi.

I diagrammi⁴ sono:

Diagramma dei casi d'uso → Mostra le relazioni tra i casi d'uso, le altre entità semantiche e gli attori. Vengono quindi mostrati insieme gli attori e i casi d'uso con le relazioni che intercorrono tra di loro. In generale questo diagramma mostra le funzionalità del sistema e le possibili interazioni con l'utente.

Diagramma di classe → E' un grafico degli elementi *classificatori*⁵ con-

⁴Per questa trattazione ci si riferisce a [2], il quale rappresenta la specifica UML 1.5. Attualmente è in fase di sviluppo UML 2.0. Ulteriori informazioni possono essere trovate all'indirizzo www.uml.org.

⁵Un classificatore è un metamodello per una classe, un tipo di dato o una interfaccia. Tutti questi modelli sono sintatticamente simili e sono rappresentati graficamente con un rettangolo contenente una parola chiave se necessario.

nessi attraverso relazioni statiche. Un diagramma di classe⁶ può anche contenere interfacce, package, relazioni così come istanze di oggetti e collegamenti.

Una caso particolare di questo tipo di diagramma è il *diagramma degli oggetti*, il quale rappresenta sia istanze di oggetti che di valori dei dati. Un diagramma degli oggetti è praticamente un'istanza di un diagramma di classe. È usato soprattutto per rappresentare lo stato di un sistema in un determinato istante temporale, ma non è molto diffuso.

Diagramma di stato → Descrive il comportamento di entità capaci di evoluzioni dinamiche in risposta alla ricezione di eventi.

Solitamente si usa questo diagramma per descrivere il comportamento di istanze di classi, ma possono essere anche usati per descrivere quello di altre entità, come casi d'uso, attori, sottosistemi, operazioni o metodi.

Diagrammi d'attività → È una variazione di una macchina a stati finiti, dove gli stati rappresentano l'esecuzione di una azione o di una sottoattività, mentre le transizioni sono innescate dal completamento di un'azione o una sottoattività. Viene rappresentata quindi la macchina a stati dell'esecuzione di una procedura.

Questo diagramma è di fatto un caso speciale di un diagramma di stato. L'intero diagramma d'attività è solitamente collegato ad un classificatore, un caso d'uso, un package o all'implementazione di una operazione.

⁶Un nome migliore per questo diagramma avrebbe potuto essere “diagramma della struttura statica”, ma “diagramma di classi” era più corto e per questo è stato scelto ([2], pag. 3-34).

I diagrammi d'attività vengono usati prevalentemente quando tutti gli eventi, o la maggior parte, sono il risultato del completamento di operazioni interne (cioè un flusso di esecuzione procedurale). I diagrammi di stato invece vengono usati quando possono verificarsi eventi asincroni (e non c'è quindi un flusso di controllo procedurale).

Diagramma di deployment, Diagramma dei componenti → Questi due diagrammi fanno parte della famiglia dei *diagrammi d'implementazione* e mostrano sia l'*implementazione fisica* del sistema che quella dei componenti e della loro dislocazione.

Le due forme di diagrammi mostrano aspetti diversi dell'implementazione: il *diagramma dei componenti* mostra la struttura dei componenti, includendo i classificatori e gli elementi che li implementano; il *diagramma dei deployment* mostra i nodi e la loro dislocazione⁷.

Diagramma di sequenza, Diagramma di collaborazione → Il diagramma di sequenza mostra l'insieme degli stimoli e i collegamenti tra le entità necessari per il completamento di una operazione. Questo tipo di diagramma può rappresentare sia classificatori (quindi strutture statiche) sia istanze di classificatori (quindi lo stato del sistema ad un determinato istante)⁸.

Questo diagramma fa parte della famiglia dei *diagrammi d'interazione*, di cui fa parte anche il *diagramma di collaborazione*, che mostra i ruoli e le connessioni tra le parti necessarie al compimento di un'operazione. Così come il diagramma dei deployment e dei componenti erano complementari tra loro, così lo sono il diagramma di sequenza e

⁷Questi due diagrammi potrebbero anche essere utili nel mostrare come in una azienda si realizza un business, dove i componenti rappresentano le procedure ed i prodotti mentre i nodi mostrano i gruppi di lavoro e le risorse (umane o di altro tipo) utilizzate.

⁸Per una trattazione più esaustiva di questo diagramma, rimando al paragrafo 1.3.

di collaborazione, che danno rispettivamente una visione temporale e spaziale della stessa interazione.

1.3 Diagramma di sequenza

1.3.1 Semantica

Un *diagramma di sequenza* rappresenta un'interazione, cioè una sequenza di messaggi scambiati tra classificatori o istanze di classificatori che hanno un ruolo all'interno di una collaborazione. Il diagramma può rappresentare un'interazione in forma generica, o una sua istanza particolare, cioè una particolare sequenza di messaggi tra quelle possibili. In entrambi i casi la collaborazione è definita dal gruppo di stimoli, tra classificatori o istanze di classificatori, necessario per eseguire una operazione o ottenere un risultato.

Un diagramma di sequenza è composto quindi da insieme una collezione di *elementi di modello* che rappresentano gli oggetti e dalle frecce che schematizzano gli stimoli. Ogni freccia ha un *sender* ed un *receiver*: il verso indica i ruoli. I diversi tipi grafici di frecce possibili (illustrati in 1.3.6) specificeranno il carattere dell'associazione. Allo stesso modo, per ciascun tipo di classificatore, esisterà una sintassi grafica differente.

1.3.2 Notazione

Un diagramma di sequenza ha due dimensioni: quella verticale e quella orizzontale. La dimensione verticale normalmente rappresenta il tempo, mentre su quella orizzontale vengono raffigurate i differenti classificatori che fanno parte della collaborazione; è possibile invertire questa convenzione, scambiando gli assi. Nella maggior parte dei casi solo la sequenza temporale degli avvenimenti è importante, ma in applicazioni real-time l'asse temporale può avere una sua metrica. In questo caso si esprimono i vincoli quotando il grafico mediante espressioni temporali o particolari etichette.

1.3.3 Opzioni di rappresentazione

L'ordine con cui i classificatori vengono disposti non è importante ma spesso, per mantenere il grafico comprensibile, le frecce procedono in una sola direzione della pagina (ad es. da sinistra a destra). Rispettare questa convenzione non è sempre possibile, quindi l'ordine dei classificatori in generale non porta informazione.

Gli assi possono essere scambiati, in modo tale da portare il tempo sull'asse orizzontale, con verso crescente da sinistra a destra e i differenti oggetti disegnati come linee orizzontali.

Varie etichette (costanti temporali, descrizione delle azioni durante una attivazione o altre informazioni utili) possono essere mostrate sia ai margini, che nei pressi della transizione o della attivazione cui si riferiscono.

Le costanti temporali possono essere indicate tramite espressioni temporali sui messaggi o sul nome degli stimoli. L'insieme delle funzioni temporali è aperto, quindi gli utenti possono inventarne di nuovi in base alle esigenze di situazioni specifiche o per distinzioni implementative.

1.3.4 Linea di vita degli oggetti

Una linea di vita denota una istanza che gioca uno specifico ruolo. Eventuali frecce tra le linee di vita denotano comunicazioni attraverso tali istanze. Da un diagramma di sequenza risultano evidenti l'esistenza e la durata di una istanza in un ruolo, mentre non lo è la relazione tra le istanze. Il ruolo viene esplicitato dal classificatore: questo mostra le proprietà dell'istanza che gioca quel ruolo e descrive la relazione che ha nei confronti delle altre.

Una linea di vita è schematizzata come una linea verticale tratteggiata e indica l'esistenza di un'istanza ad un determinato tempo. Se una istanza viene creata o distrutta durante un periodo di tempo mostrato dal diagramma, allora la sua linea di vita partirà e/o finirà nel punto appropriato; altrimenti la linea attraverserà tutto il diagramma dall'alto verso il basso. Il simbolo

di un oggetto è mostrato alla testa della linea di vita. Se l'istanza è creata nel corso dell'interazione, una freccia (che mappa lo stimolo che genera la nascita dell'oggetto) è disegnata con la testa verso l'oggetto. Se l'istanza è distrutta durante la vita del diagramma, una grossa "X" indica tale distruzione, associata ad uno stimolo, che può essere la causa della distruzione o il valore restituito dall'elaborazione compiuta dall'oggetto. Un'istanza già viva quando la transazione inizia è mostrata in alto nel diagramma, mentre un'istanza ancora viva quando la transazione finisce ha la sua linea di vita che continua oltre la freccia finale.

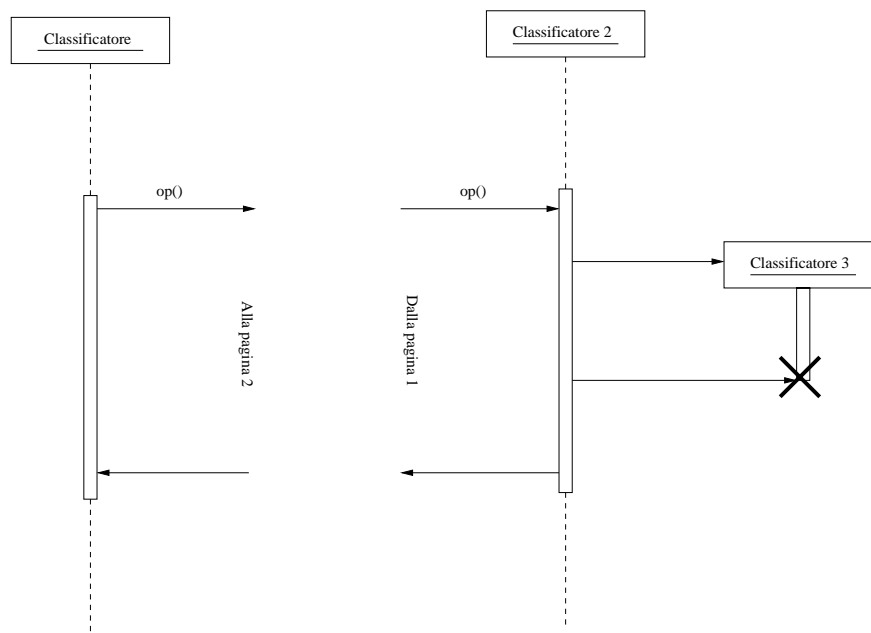


Figura 1.1: Esempi grafici di linee di vita

Una linea di vita può essere spezzata in una o più linee di vita concorrenti per mostrare comportamenti condizionali. Ogni traccia separata corrisponde ad una via di comunicazione. Le linee di vita potranno poi ricongiungersi in seguito.

In alcuni casi è necessario collegare tra loro più diagrammi di sequenza. Questo potrebbe essere necessario ad esempio quando l'intera linea di vita

non può essere contenuta in un unico diagramma. In questo caso è possibile collegare tra loro più diagrammi, etichettando opportunamente stimoli “pendenti” in uscita da un diagramma ed i corrispondenti stimoli “pendenti” in entrata su un altro diagramma. Ovviamente, per rendere evidente questa scelta, sarà necessario etichettare opportunamente i due diagrammi.

Tutte le varianti descritte sono mostrate nelle diverse parti della figura 1.1.

1.3.5 Attivazione

Un’attivazione evidenzia il periodo durante il quale un’istanza sta eseguendo una procedura sia direttamente, sia attraverso procedure subordinate, mostrando contemporaneamente la durata dell’operazione nel tempo e la relazione di controllo tra l’attivatore e i suoi attivati.

Graficamente un’attivazione è mostrata come un rettangolo sottile nel quale il lato superiore è allineato all’istante stesso di attivazione, mentre il lato inferiore è allineato all’istante di completamento. La procedura in corso di esecuzione può essere etichettata con un testo opportuno in corrispondenza dell’attivazione. Alternativamente la procedura può essere segnalata dalle frecce in arrivo: in tal caso il testo di attivazione può essere omesso.

Le notazioni menzionate sono visibili nella figura 1.1, dove alcune frecce sono anonime, mentre altre portano il nome della procedura.

1.3.6 Messaggi e stimoli

Uno stimolo è una comunicazione tra due istanze: questo può causare la chiamata di un’operazione, l’emissione di un segnale o la creazione/distruzione di una istanza. Un messaggio è la specifica di uno stimolo, in quanto descrive a quali ruoli il sender e il receiver devono attenersi per soddisfare uno stimolo conforme al messaggio.

In un diagramma di sequenza stimoli e messaggi sono mostrati come

freccie orizzontali (graficamente diverse a seconda del messaggio) da una istanza ad un'altra. In caso di chiamata a se stesso, la freccia parte e finisce dalla stessa linea di vita. La freccia è etichettata con il nome dell'operazione da invocare, il nome del segnale o il valore di ritorno. Gli argomenti possono esserci, ma non sono obbligatori. La freccia potrebbe essere etichettata con un numero di sequenza univoco, che consenta di scandire la giusta sequenza temporale delle chiamate. Questa possibilità rimane però spesso inutilizzata, in quanto la sequenza temporale è anche definita graficamente nei diagrammi di sequenza. Può essere comunque utile numerare le chiamate per metterle in relazione con quelle di un eventuale diagramma di collaborazione associato.

Le frecce hanno diverse varianti grafiche per differenziare il tipo delle comunicazioni. Queste possono essere:

freccia piena/riga continua → Utilizzata per le chiamate di operazioni o per altri tipi di controllo del flusso annidato. L'intera sequenza di operazioni annidate è completa quando il livello più esterno è completo.

freccia vuota/riga solida → Comunicazioni asincrone: non c'è nessun annidamento del controllo, poichè il sender manda lo stimolo e subito dopo ritorna all'istruzione successiva della sua esecuzione.

freccia vuota/riga tratteggiata → Ritorno da una chiamata di operazione.

Le varie tipologie di freccia sono disegnate nella figura 1.2.

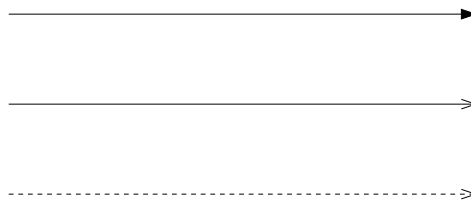


Figura 1.2: Esempi grafici di messaggi e stimoli

Alcune varianti del diagramma meritano di essere elencate:

- In una sequenza di messaggi procedurali, il ritorno non è necessario. Lo è invece in presenza di messaggi o stimoli asincroni;
- I messaggi sono disegnati normalmente con linee orizzontali. Questa notazione presuppone che la chiamata venga ricevuta istantaneamente. Tale ipotesi può considerarsi vera sui calcolatori. Se lo stimolo richiede un tempo non trascurabile per arrivare dal sender al receiver, la freccia sarà piegata verso il basso, in modo che la partenza avvenga ad un'istante antecedente all'arrivo: questa notazione è necessaria nel caso in cui possano giungere nuovi messaggi nel tempo che intercorre tra la partenza e l'arrivo.
- Da un singolo punto può partire più d'una freccia. Ciò può essere necessario nel caso che si possa inviare più di un messaggio in base a determinate condizioni.
- La ricorsione viene indicata con una linea di vita affiancata a quella principale, una freccia d'ingresso ed una di ritorno. Opzionalmente può essere inserita una etichetta.
- Una linea di vita può fare le veci di un gruppo di oggetti, rappresentando quindi una visione ad alto livello.
- E' formalmente utile distinguere tra un periodo in cui l'istanza è attiva, ma in attesa di un evento e uno in cui l'istanza sta effettivamente svolgendo un'elaborazione. La prima viene rappresentata con la doppia linea, la seconda può essere evidenziata da una regione ombreggiata nella linea di vita.

1.3.7 Transizioni temporali

Un messaggio può specificare differenti istanti: ad esempio uno di spedizione ed uno d'arrivo. Ci sono nomi formali che possono essere usati per descrivere le espressioni che denotano tali istanti. Il set dei differenti tipi di istanti temporali è aperto, così che gli utenti possono inventarne di nuovi per situazioni particolari. Ad esempio, se il nome dello stimolo è “*stim*”, il suo istante di partenza è espresso come “*stim.sendTime()*”.

Una costante temporale è un'espressione che si ricava a partire dal nome della transizione. Le costanti temporali sono mostrate sul margine sinistro, allineate alla freccia.

Capitolo 2

Librerie di supporto

2.1 GTKmm

2.1.1 Perché GTKmm?

GTK+ è attualmente uno tra i più evoluti toolkit¹ grafici e, per taluni aspetti, anche innovativo. Poiché per questo progetto si è scelto di lavorare con tecniche orientate agli oggetti facendo uso del linguaggio C++, le GTK+, scritte in C, non potevano andare bene per la realizzazione dell'interfaccia grafica per l'utente (GUI). Volendo comunque rimanere nell'ambito Open Source, le possibilità erano due: il wrapper delle GTK+ in C++ (GTKmm) o le librerie Qt. La scelta è caduta sulle GTKmm soprattutto per via del linguaggio di programmazione: le Qt, scritte in un periodo in cui il C++ non era ancora stato standardizzato, necessitano di una forma di scrittura del codice un po' particolare (legate alle scelte di progetto del team di sviluppo delle Qt) e di un preprocessore aggiuntivo (MOC), oltre a quello standard,

¹Maggiori informazioni sul toolkit GTK+ possono essere trovate all'indirizzo <http://www.gtk.org/>.

per rispettare a tempo di compilazione le regole del linguaggio². Le GTKmm invece sono state da subito scritte rispettando lo standard ANSI e sfruttandolo al massimo per ottenere le migliori prestazioni e un maggiore supporto dal controllo di tipo. Non solo, le GTKmm utilizzano pesantemente la STL (Standard Template Library) per ottenere un'interfaccia alla libreria che il programmatore esperto già sicuramente conosce. Si rimanda a 2.1.2 e [4] per ulteriori particolari sulle caratteristiche di GTKmm.

2.1.2 Come creare le interfacce con le GTKmm

Le GTKmm utilizzano il *widget packing system*. Diversamente dalla maggior parte dei toolkit per la realizzazione di interfacce grafiche, che adottano un sistema di posizionamento degli elementi grafici di tipo assoluto, le GTKmm adottano un sistema incastonato: ogni widget può contenere un altro widget. Questo permette all'interfaccia grafica di essere "elastica".

In particolare, il posizionamento assoluto presenta i seguenti problemi:

- I widget non si riposizionano automaticamente quando la finestra viene ridimensionata. Questo può nascondere alcuni widget se la finestra è troppo piccola oppure presentare dello spazio inutilizzato se la finestra è troppo grande;
- È impossibile predire la dimensione del testo dopo una modifica o dopo il cambio di dimensione dei font. In particolare, su Unix, è impossibile anche predire gli effetti di ciascun tema o del window manager;
- Calcolare il layout della finestra "al volo" per aggiungere nuovi widget può risultare difficoltoso: è necessario in questo caso ricalcolare la posizione di tutti i widget;

²Maggiori informazioni sul toolkit Qt possono essere trovate all'indirizzo <http://www.trolltech.com/>.

GTKmm ha il vantaggio di non presentare questi problemi. Invece che specificare la posizione e la dimensione di ogni widget, GTKmm ordina i widget in righe, colonne e/o tabelle. Sarà poi compito di GTKmm disporre tutti gli elementi e calcolarne la giusta dimensione nella finestra in base agli altri widget contenuti. La dimensione dei widget viene calcolata tenendo conto del testo che devono contenere, delle dimensioni min e max specificate e del modo in cui deve essere diviso lo spazio comune. Il layout può essere poi perfezionato specificando il *padding*³ ed il valore centrale per ogni widget. GTKmm usa tutte queste informazioni per ridimensionare e riposizionare i widget in funzione del contesto e dell'interazione con l'utente. Per certi versi, questo stile di visualizzazione è analogo al rendering grafico delle pagine web: anche in quel caso la disposizione degli elementi è “al volo”, adattativa rispetto al contesto e realizzata tramite strutture contenitore/contenuto⁴.

La disposizione degli elementi è quindi *gerarchica*. A tal proposito GTKmm definisce i *contenitori*, i quali possono contenere altri contenitori oppure widget.

I contenitori si dividono in due gruppi:

A figlio singolo → Possono contenere un solo contenitore o un solo widget. Derivati da `Gtk::Bin`.

A figlio multiplo → Possono contenere più di un contenitore e/o widget. Derivati da `Gtk::Container`.

Nella figura 2.1 si vede come da `Gtk::Container` vengono derivati i vari container. Vengono poi mostrati quelli più significativi. In particolare, si vede come `Gtk::Window` è un container a figlio unico, anche se questo potrebbe

³Letteralmente “riempimento”. In questo contesto indica l'ampiezza del bordo tra il widget ed il suo contenitore.

⁴Nel caso delle pagine web i contenitori possono essere tabelle, layer o frame, mentre il contenuto è essenzialmente costituito da testo o immagini.

sembrare contro la logica. In pratica questa soluzione diventa molto flessibile nelle mani del programmatore, che può decidere di sistemare i widget all'interno di una finestra nel modo che ritiene più opportuno, incastrando i vari container senza schemi prefefiniti.

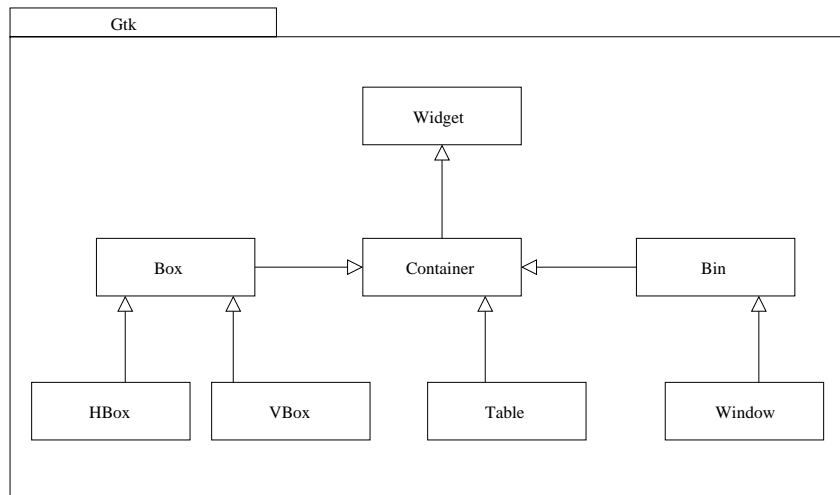


Figura 2.1: Parte della derivazione da `Gtk::Container`

Il widget packing system è un'implementazione del pattern decorator [3], la cui struttura è rappresentata in figura 2.2. Il pattern decorator, nella sua forma più generale, permette di includere il componente da decorare in un altro, responsabile dell'aggiunta della decorazione. L'oggetto contenitore è chiamato decorator. Grazie alla trasparenza è possibile annidare i decoratori in modo ricorsivo, consentendo così agli oggetti decorati di aver un numero potenzialmente illimitato di responsabilità.

Seguendo la teoria dei design patterns - e del pattern decorator in particolare - le interfacce si compongono praticamente secondo i seguenti passi:

→ Scegliere il widget desiderato ed istanziarlo come variabile membro di

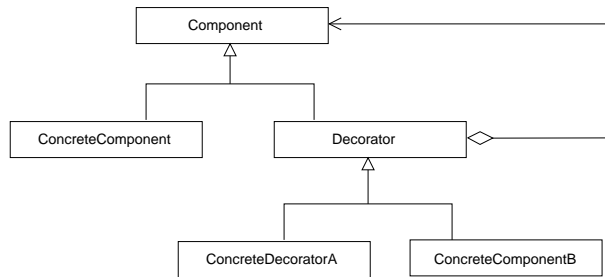


Figura 2.2: Pattern Decorator

una classe o utilizzarlo come base di una derivazione⁵;

- Impostare gli attributi dei widget⁶;
- Connettere tutti i segnali che si vogliono usare con i rispettivi handler (vedi sezione 2.2.2);
- Incastonare il widget dentro il suo container, utilizzando le funzioni tipiche della STL (come discusso nella sezione 2.1.1);
- Chiamare `show()` sul widget, per disegnarlo;

È importante sottolineare che in questo ultimo passo, la chiamata della funzione `show()` sull'oggetto radice dell'albero dei widget è obbligatoria, mentre potrebbe essere omessa per alcuni oggetti all'interno dell'albero al fine di ottenere particolari incastri dei widget (ad es. la comparsa di una list-box in seguito al click di un bottone). Ciò permette una maggiore interattività tra interfaccia e l'utente. Se invece si vuole chiamare la `show` su

⁵È anche possibile l'utilizzo di puntatori. In tal caso è molto utile tenere il puntatore come membro di una classe in modo da poter chiamare in fase di esecuzione i metodi che soddisfano le richieste dell'utente.

⁶Se un contenitore è utilizzato come classe base e non possiede costruttore default, così come definito dal C++, la classe base deve essere costruita nella lista di inizializzazione del costruttore della classe derivata.

tutti i widget, si può chiamare la `show_all` sul widget padre, che si farà carico di chiamare la `show` su tutti gli elementi dell'albero.

2.1.3 Un esempio di interfaccia

Per evidenziare praticamente il funzionamento di GTKmm e del widget packing system, nei listati 2.1 e 2.2 sono mostrati rispettivamente la definizione della classe `MainWindow` ed il suo costruttore.

```
//  
// File: MainWindow.h  
// Created by: Davide "Rocker" Anastasia <s242248@studenti.ing.unipi.it>  
// Created on: Sat May 1 13:29:11 2004  
//  
  
#ifndef _MAINWINDOW_H_  
#define _MAINWINDOW_H_  
  
#include <gtkmm.h>  
#include <libgnomecanvasmm.h>  
  
#include "CanvasMgr.h"  
#include "ModelClassObject.h"  
  
class MainWindow : public Gtk::Window {  
    public:  
        MainWindow();  
        virtual ~MainWindow();  
  
        Gnome::Canvas::Canvas* getCanvas();  
  
    protected:  
        // Container principale  
        Gtk::VBox* vbox_;  
  
        // Barra dei menu  
        Gtk::MenuBar* menubar_;  
  
        // Bottoni in alto...
```

```
        Gtk::Toolbar* handle_b_box_;\n\n        // Canvas e scrollwindow\n        Gtk::ScrolledWindow* scrollwindow_;\n        CanvasMgr* canvas_;\n\n        // Status Bar\n        Gtk::Statusbar* status_;\n};\n\n#endif // _MAINWINDOW_H_
```

Listato 2.1: Definizione di MainWindow

Come si vede, nella zona *protected* della classe vengono definiti una serie di puntatori ad oggetti del namespace `Gtk`. Tali membri dato conterranno quindi il puntatore ad un oggetto che comporrà l'interfaccia grafica.

Il numero di puntatori presenti in questa classe è ridotto perché si è scelto di dividere le varie parti dell'interfaccia in sottoclassi realizzate tramite derivazione da classi delle GTKmm. Questo concetto sarà più chiaro esaminando il codice del costruttore.

```
//\n// File: MainWindow.cc\n// Created by: Davide "Rocker" Anastasia <s242248@studenti.ing.unipi.it>\n// Created on: Sat May 1 13:29:11 2004\n//\n#include <gtkmm.h>\n#include <libgnomecanvasmm.h>\n\n#include "MainWindow.h"\n#include "ButtonBar.h"\n#include "MyMenuBar.h"\n#include "CanvasMgr.h"\n#include "ModelClassObject.h"
```

```

#include "ModelUser.h"
#include "ModelRecursion.h"
#include "ModelMessages.h"
#include "ModelBornd.h"

MainWindow::MainWindow()
{
    // Status Bar
    status_ = manage( new Gtk::Statusbar() );

    // Istanziamento del canvas e dello scrollwindow e settaggio delle propriet
    canvas_ = manage( new CanvasMgr(status_) );
    canvas_>set_size_request(CANVAS_X/2, CANVAS_Y/2);
    canvas_>set_scroll_region(0, 0, CANVAS_X, CANVAS_Y);

    canvas_>set_flags(Gtk::CAN_FOCUS);
    canvas_>grab_focus();

    // La scrollwindow permette di visualizzare una parte del canvas
    scrollwindow_ = manage(new Gtk::ScrolledWindow);
    scrollwindow_>add(*canvas_);
    scrollwindow_>set_hadjustment(*canvas_>get_hadjustment());
    scrollwindow_>set_vadjustment(*canvas_>get_vadjustment());
    scrollwindow_>get_hscrollbar()>set_increments(20.0, 50.0);
    scrollwindow_>get_vscrollbar()>set_increments(20.0, 50.0);

    // Creazione della HandleButtonBox
    handle_b_box_ = manage( new ButtonBar(canvas_) );

    // Creazione del men in alto
    menubar_ = manage( new MyMenuBar(canvas_, this) );

    // Inizializzazione del widget principale
    vbox_ = manage( new Gtk::VBox() );

    // Riempimento del widget principale
    vbox_>pack_start(*menubar_, 0, 0);
    vbox_>pack_start(*handle_b_box_, 0, 0);
    vbox_>pack_start(*scrollwindow_, Gtk::PACK_EXPAND_WIDGET, 0);
    vbox_>pack_start(*status_, 0, 0);

    // Aggiunta del widget principale alla finestra

```

```
    add(*vbox_);  
    show_all();  
}
```

Listato 2.2: Costruttore di `MainWindow`

Dai precedenti spezzoni di codice, si deducono alcuni concetti interessanti:

- La funzione `manage()` prende come argomento un puntatore ad un oggetto delle GTKmm e restituisce un puntatore “intelligente”. L’oggetto restituito è uno *Smart Pointer*, cioè un oggetto che si comporta come un puntatore, ma permette una migliore gestione della memoria, in quanto la libera se non esiste più nessun puntatore che lo riferisce (si adotta una soluzione basata sul *reference counting*). Questa funzionalità si ottiene tramite una classe speciale a cui è stato ridefinito `operator->`⁷.
- La funzione `add(Widget& widget)` prende come argomento un riferimento ad un `Gtk::Widget` e restituisce `void`. Rappresenta la funzione d’interfaccia in grado di includere un oggetto dentro un container a figlio unico.
- La funzione `pack_start(...)`, presente in più varianti a seconda del tipo degli argomenti, rappresenta la funzione d’interfaccia capace di includere un oggetto dentro un container a figlio multiplo. La posizione viene calcolata in base all’ordine procedurale con cui i figli vengono agganciati al padre: ad esempio, il terzo `pack_start()` inserirà il figlio in terza posizione. Questa funzione è molto simile alle `push_back()` e `push_front()` presenti sia in alcuni tipi di strutture dati della STL per l’inserzione, sia nelle GTKmm. Infatti alcuni tipi di container

⁷Per maggiori informazioni sugli *Smart Pointer*, si rimanda alla sezione 11.10 a pagina 326 di [7] ed alla sezione riguardante `RefPtr` di [6].

sono derivati dalle classi che implementano queste strutture dati e ne ereditano quindi l'interfaccia.

2.2 SigC++

I programmi “a finestre” sono per loro intrinseca caratteristica *event-driven*, cioè guidati dalle interazioni tra l'interfaccia grafica e l'utente. Questo tipo d'interazione è ovviamente asincrona, in quanto in nessun modo è possibile stabilire quando l'utente premerà un bottone, piuttosto che selezionare una voce di menù: l'utente tramite uno strumento di input, come la tastiera o il mouse, “eccita” l'interfaccia grafica, che in risposta ad un evento esegue una determinata operazione. Questo procedimento viene realizzato per mezzo dei segnali: viene cioè operata una separazione tra chi genera l'evento e chi lo gestisce.

2.2.1 Perché SigC++?

SigC++ é una libreria C++ *type safe* espressamente realizzata per lavorare in comunione con GTKmm, ma che non si limita a questo solo utilizzo. Essendo realizzata per rispettare tutti gli standard del C++, motivo per cui è anche portabile, può infatti fare da framework per la gestione dei segnali in qualsiasi contesto, compresi i programmi realizzati con API Win32.

SigC++ fornisce il concetto di *slot*, una struttura dati contenente il riferimento ad un *callback*⁸. Come callback possono essere utilizzati:

→ **Funzioni globali, statiche o friend;**

⁸Callback è sinonimo di *handler*, ed in questa ottica deve essere inteso: come gestore dell'evento a cui è associato.

- **Puntatori ad oggetti** che ridefiniscono `operator()`⁹;
- **Puntatori a membro e istanze di oggetto** su cui invocare l'operazione (l'oggetto deve essere derivato da `SigC::Object`¹⁰).

Ognuna delle possibilità elencate può prendere più argomenti di tipo differente. Per facilitare la costruzione di uno slot, SigC++ fornisce un template overloaded chiamato `Slot` che può prendere un argomento `o`, se necessario, più d'uno e restituire uno `Slot` generico che può essere invocato tramite `operator()`.

SigC++ fornisce inoltre un oggetto `Signal`, al quale il client (fondamentalmente il programmatore) può collegare alcuni `Slot`. Ogni volta che un segnale viene emesso, tutti gli `Slot` ad esso collegati vengono richiamati.

Riassumendo, la gestione dei segnali è divisa in tre parti:

- Il chiamante
- Il ricevitore
- Una connessione tra chiamante e ricevitore

È importante notare che la relazione tra chiamante e ricevitore è di tipo 1 a n, cioè un chiamante può invocare più handler.

2.2.2 Come gestire i segnali

Negli esempi che seguono vengono mostrate le diverse possibilità offerte da SigC++. In particolare verrà evidenziato come è possibile collegare i

⁹Per maggiori informazioni, si consulti il paragrafo 2.4.4 e [7] al capitolo sugli *Oggetti Funzione*.

¹⁰Ogni singolo widget delle GTKmm ha come superclasse `SigC::Object` e questo, nel caso di creazione di widget personalizzati tramite derivazione da classi delle GTKmm, ha il vantaggio di non rendere esplicita la derivazione da `SigC::Object`.

segnali con gli handler¹¹.

```
#include <iostream>
#include <sigc++/sigc++.h>
#include <unistd.h> // unistd.h only needed for 'sleep()' used in AlienDetector::run()

using namespace std;

class AlienDetector
{
public:
    AlienDetector() {}
    void run();

    SigC::Signal0<void> detected;
};

void
warn_people()
{
    cout << "There are aliens in the carpark!" << endl;
}

int
main()
{
    AlienDetector mydetector;
    mydetector.detected.connect( SigC::slot(warn_people) );

    mydetector.run();

    return 0;
}

void
AlienDetector::run()
{
    sleep(3);
}
```

¹¹Gli esempi sono tratti dalla documentazione del pacchetto SigC++

```
    detected.emit();  
}
```

Listato 2.3: Utilizzo di SigC++: collegamento con una funzione globale

Il listato 2.3 mostra com'è possibile collegare un segnale con una funzione globale. Le righe interessanti da esaminare sono due:

Riga 13 → il membro della classe definito è un segnale, cioè un oggetto funzione che si fa carico di scatenare l'handler in determinate circostanze. La chiamata dell'handler, per le proprietà degli oggetti funzione, avviene come una normale chiamata a funzione;

Riga 26 → viene effettuato il collegamento del segnale con il ricevitore, la funzione globale `warn_people()`. La connessione avviene tramite la `connect()`, la quale prende come parametro uno `Slot` che si farà poi carico di trasportare il segnale dal chiamante al ricevitore.

```
#include <iostream>  
#include <string>  
#include <sigc++/sigc++.h>  
#include <unistd.h>  
  
using namespace std;  
  
class AlienDetector  
{  
public:  
    AlienDetector() {}  
    void run();  
    SigC::Signal0<void> detected;  
};  
  
class AlienAlerter : public SigC::Object  
{  
public:  
    AlienAlerter(char const* servername) : servername_(servername) {}  
    void alert();  
private:
```

```

        std::string servername_;
    };

    void
    AlienAlerter::alert()
    {
        cout << "Message from " << servername_ << ": Aliens have landed in the carpark!"
              << endl;
    }

    int
    main()
    {
        AlienDetector mydetector;
        AlienAlerter myalerter("localhost");
        mydetector.detected.connect( SigC::slot(myalerter, &AlienAlerter::alert) );

        mydetector.run();

        return 0;
    }

    void
    AlienDetector::run()
    {
        sleep(3);
        detected.emit();
    }

```

Listato 2.4: Utilizzo di SigC++: collegamento con una funzione membro

Il listato 2.4 mostra il collegamento di una funzione membro. Diversamente dall'esempio precedente, lo slot prende come parametri l'oggetto (`myalerter`) e la funzione da invocare su di esso (`&AlienAlerter::alert`). Lo Slot generato viene comunque passato alla funzione `connect()` (riga 36) in modo perfettamente analogo. È possibile (e spesso molto utile) passare come parametro `this`.

Tra le altre capacità di SigC++ si possono elencare:

→ passaggio di parametri alle funzioni handler (*rebinding*);

- restituzione da parte della funzione handler di valori con tipo diverso da `void`;
- controllo del processo di emissione di un segnale e possibilità di fermare la corsa del segnale stesso, bloccandone il processo di emissione nel caso in cui l'evento sia stato gestito correttamente (*marshalling*)¹²;
- ridefinizione del tipo dell'handler (*retyping*).

2.3 Libgnomecanvasmm

“*Canvas*” è il nome tecnico dell'area di disegno di una applicazione. `Libgnomecanvasmm` è il wrapper C++ del `libgnomecanvas` scritto in C. Questa libreria permette di gestire ad oggetti l'area di disegno: ogni elemento disegnato (rettangoli, spezzate, punti, cerchi, ecc.) è realizzato all'interno dell'applicazione come un oggetto, permettendo quindi una flessibilità e un'astrazione maggiore.

L'alternativa al canvas è la `DrawingArea` delle GTK+ (in C) o delle GTKmm (in C++). Questa alternativa è stata subito scartata per motivi pratici, come la difficoltà di gestire primitive geometriche in maniera astratta (ogni forma deve essere disegnata come una serie di punti) e di catturare gli eventi in maniera semplice (`libgnomecanvasmm` mette già a di-

¹²Il *marshalling* è certamente la funzione più complessa e potente della libreria. Ogni segnale possiede un *Marshaller* default, il quale si limita semplicemente a non far nulla. Se questo *Marshaller* viene ridefinito, seguendo lo scheletro stabilito dalla libreria, è possibile raccogliere tutti i valori restituiti da un segnale (poiché ad un segnale possono essere collegati più handler, un segnale può restituire più valori) ed effettuarne una elaborazione. La classe *Marshaller*, tramite il suo metodo `value()`, è poi in grado di restituire il risultato di questa elaborazione al chiamante del segnale.

Ovviamente, per sfruttare questa possibilità, è necessario definire il segnale in modo appropriato. Per definire i propri *Marshaller* è possibile lavorare per derivazione dai *Marshaller* predefiniti della libreria.

sposizione la gestione degli eventi, anche se in maniera differente da quanto fa GTKmm). Gli sviluppatori della DrawingArea di GTKmm definiscono questo widget come la base per la creazione di widget specializzati in particolari applicazioni.

Libgnomecanvasmm può essere reperito all'indirizzo sul sito ufficiale delle GTKmm [4], come tutta la serie "mm".

Il canvas, prima di poter essere istanziato ed utilizzato, deve essere inizializzato, invocando `Gnome::Canvas::init()`. Questa funzione ausiliaria assegna i valori iniziali alle strutture dati usate internamente dal canvas e invisibili al programmatore.

Il canvas può essere di due tipi:

Standard → È rappresentato da una istanza di `Gnome::Canvas::Canvas` ed è il wrapper C++ dello Gnomecanvas scritto in C.

AntiAliased → È un'istanza di `Gnome::Canvas::CanvasAA` ed ha il vantaggio rispetto al precedente di supportare l'*antialiasing*.

L'*antialiasing* è una caratteristica derivata dalle GTK+ 2.x: permette il rendering di testo, widget e disegni in maniera tale che abbiano un'aspetto particolarmente gradevole nella visualizzazione. Le tecniche di *antialiasing* permettono di "sfumare" gli elementi da visualizzare aggiungendo ad essi un alone.

Per la realizzazione di questo progetto si è scelto di usare il canvas "standard", poiché l'*antialiasing* è una feature ancora in via di sviluppo e può creare problemi nella visualizzazione di alcune proprietà (come le linee tratteggiate).

2.3.1 La gestione degli eventi

Le GTKmm permettono la gestione degli eventi attraverso segnali associati a ciascuno di essi (ad es. *on_focus*, *on_clicked*, *on_grab*). Il canvas

invece genera un segnale generico, il cui evento scatenante è individuato da una struttura dati (`GdkEvent`) inizializzata con valori opportuni. Questa viene passata come argomento all'handler del segnale che, in relazione ai valori presenti all'interno della struttura, decide qual è la giusta sequenza di operazioni da compiere. Per questo motivo si possono avere handler molto grandi, composti da una serie di switch, anche annidati.

```
bool
AnchorBox::on_event(GdkEvent* event)
{
    Gnome::Canvas::Ellipse::on_event(event);

    switch(event->type) {
        case GDK_ENTER_NOTIFY:
            property_fill_color() = "yellow"; /*Definizione di una proprieta */
            break;
        case GDK_LEAVE_NOTIFY:
            if(!(event->crossing.state & GDK_BUTTON1_MASK)) {
                property_fill_color() = "green";
            }
            break;
        case GDK_BUTTON_PRESS:
            grab(GDK_POINTER_MOTION_MASK | GDK_BUTTON_RELEASE_MASK,
                Gdk::Cursor(Gdk::FLEUR),
                event->button.time);
            break;
        case GDK_BUTTON_RELEASE:
            ungrab(event->button.time);
            break;
        default:
            break;
    }
    return false;
}
```

Listato 2.5: Gestore di segnale default della classe `AnchorBox`

Il listato 2.5 mostra una porzione di codice utilizzata spesso nel progetto: è un gestore di segnale default delle primitive del canvas. Ogni elemento primitivo del canvas è legato (senza il bisogno che questo venga esplicitato con `SigC++`) ad un gestore default (prototipato come `bool on_event(GdkEvent*)`). Ridefinendolo è possibile compiere operazioni ausiliarie: in particolare è stato usato questo gestore nelle classi `AnchorBox` e `DragBox` per separare il cambiamento dello stile dal comportamento desiderato (ad es. il movimento di un oggetto), che è definito tramite un segnale ad una apposita funzione. Così, mentre le proprietà grafiche sono generali, quelle comportamentali possono cambiare in quanto derivano dal contesto di esecuzione e dall'operazione richiesta.

Oltre ai gestori default, è possibile collegare alle varie primitive del canvas altri tipi di segnali, con una sintassi già vista quando si è parlato di `SigC++` (in particolare nel listato 2.4):

```
lifetime_box_->signal_event().connect(slot(*this, &ModelClassObject::eventCtrl));
```

Questa riga è tratta dal costruttore della classe `ModelClassObject` e mostra come all'oggetto `lifetime_box_`, istanza di `Gnome::Canvas::Rect` e membro dato di `ModelClassObject`, è possibile collegare la funzione `eventCtrl` facente parte dell'interfaccia della classe stessa.

2.3.2 La definizione delle proprietà degli oggetti

La riga 8 del listato 2.5 costituisce un esempio di come sia possibile definire una proprietà di un oggetto del canvas.

Il meccanismo utilizzato da `libgnomecanvasmm` si basa sul concetto di *proxy*: chiamando una funzione (come `property_fill_color()` nell'esempio) si ottiene un "riferimento" alla proprietà associata alla funzione. Il proxy può essere utilizzato per ottenere il valore della proprietà o per dare un nuovo valore. Il contesto decide qual è il compito del proxy nell'operazione richiesta. Nell'esempio in questione si pone il colore di riempimento di

un'ellisse (in realtà un cerchio) a "yellow". Quando il contesto non definisce univocamente qual è il ruolo del proxy in un'espressione, il programmatore può utilizzare i metodi `get_value` e `set_value` definiti nella classe template `Glib::PropertyProxy`.

2.4 STL - Standard Template Library

Per parlare della STL un libro potrebbe non bastare e non si ha quindi nessuna pretesa di completezza in questo contesto. Vengono forniti semplicemente alcuni cenni sugli algoritmi e le strutture dati sfruttati nella programmazione, rimandando al capitolo 3 per i particolari su come questa libreria è stata utilizzata.

2.4.1 List

`List` è una sequenza ottimizzata per l'inserzione e l'eliminazione degli elementi. Poiché `list` è una classe template, quando viene creata un'istanza è necessario specificare il tipo degli elementi in essa contenuti. Si userà quindi la forma:

```
std::list<ModelObjBase*>* childs_;
```

Come si vede da questa definizione, gli elementi della lista sono puntatori: questo metodo diventa utile quando uno stesso elemento deve essere contenuto in più liste contemporaneamente. La gestione delle strutture dati con puntatori richiede una particolare attenzione, poiché un puntatore pendente su cui viene tentata una deferenza genererà con altissima probabilità un errore a tempo di esecuzione, il quale provocherà l'uccisione del processo da parte del sistema operativo.

Per la natura stessa di `list`, l'operazione di selezione con indice, tipica dei vettori, non viene fornita. Con lo stesso principio però vengono

definite operazioni apposite di ricerca, eliminazione ed inserimento, nonché operazioni particolari sul primo elemento.

2.4.2 Iteratori

Nel caso di strutture dati aggregate (`list` ne è un esempio) dovrebbe essere possibile accedere ai vari elementi senza esporre la struttura interna. Inoltre è desiderabile poter attraversare la lista in modi diversi, in base al contesto. Una possibile soluzione è quella di aggiungere le funzioni per la gestione della lista nell'interfaccia della classe, ma ciò determinerebbe un sovraccarico ed alcune limitazioni, come l'impossibilità di attraversare la lista contemporaneamente più volte ed in modo diverso.

Il problema descritto viene risolto tramite l'utilizzo di un `Iterator`, cioè un oggetto responsabile dell'accesso e dell'attraversamento della lista. La STL fornisce molti tipi di iteratori, per diverse politiche d'accesso, offrendo libertà di implementazione al programmatore, che può scegliere in ogni situazione il metodo di accesso che ottimizza le prestazioni (ad es. in alcuni casi per scorrere una lista, può essere più redditizio partire dal fondo invece che dall'inizio).

Un iteratore è strettamente legato alla lista che scorre e la sua inizializzazione viene effettuata tramite un'istruzione come la seguente:

```
| std::list<ModelObjBase*>::iterator iter = childs_>begin();
```

Questa riga crea un oggetto iteratore `iter`, posizionando il puntatore sul primo elemento (operazione realizzata dalla funzione `begin()`). Per scorrere tra gli elementi l'iteratore ridefinisce gli operatori di decremento e incremento, rispettivamente per tornare indietro o andare avanti di un elemento. Per accedere all'elemento corrente si utilizza l'operatore di deferenza. In tal modo una riga come:

```
| delete *iter;
```

ha l'effetto di distruggere l'elemento corrente, quello cioè su cui si trova l'iteratore. Un'operazione di questo tipo è corretta se l'elemento corrente è un puntatore ed un tal controllo può essere effettuato dal compilatore.



Figura 2.3: Pattern Iterator

Tutti gli iteratori della STL possiedono la stessa interfaccia: questo permette in tempi successivi di cambiare una struttura dati senza preoccuparsi di rivedere tutto il codice relativo agli iteratori. Ciò consente di ottenere un maggior grado d'astrazione, poiché il codice relativo alla gestione degli iteratori non dipende dalla struttura dati su cui lavorano. In letteratura questo pattern viene indicato con il nome di `iterator` e la sua struttura è mostrata in figura 2.3.

2.4.3 Algoritmi standard

La STL fornisce una serie molto vasta di algoritmi per la manipolazione delle strutture dati. In questa breve trattazione verrà analizzato il funzionamento di quelli che sono stati ampiamente utilizzati nella programmazione del progetto.

`for_each`

La `for_each()` è una funzione della libreria standard che permette di eseguire una certa operazione per ogni elemento di una sequenza. È prototipata come:

```
Function for_each(ForwardIterator first, ForwardIterator
last, Function func);
```

ed è rintracciabile in `<algorithm>`.

Ognuno degli elementi all'interno del range `[first, last)` dell'iteratore è passato a turno come riferimento alla funzione (o all'oggetto funzione) `func`, la quale può modificare l'elemento che ha ricevuto. La funzione stessa (o l'oggetto funzione) viene restituito come valore finale dell'elaborazione, mentre il tipo di ritorno di `func` viene ignorato dalla `for_each()`.

Quando si ha a che fare con contenitori di puntatori può essere utile chiamare una funzione membro dell'oggetto puntato, invece che una funzione globale sul puntatore (cosa che `for_each()` farebbe nel caso standard). È quindi necessario un meccanismo di conversione, meccanismo che viene offerto dalla funzione `mem_fun()` della libreria standard. Questa funzione prende come argomento un puntatore a una funzione membro e produce qualcosa che può essere chiamato per un puntatore alla classe della funzione membro. Il meccanismo di `mem_fun()` è importante perché permette agli algoritmi standard di essere usati su contenitori di oggetti polimorfici¹³.

find_if

Gli algoritmi `find()` scorrono una sequenza cercando un valore che verifichi una determinata condizione. La `find_if()` cerca un valore che soddisfi un predicato passato come argomento. È prototipata come:

```
InputIterator find_if(InputIterator first, InputIterator
last, Predicate pred);
```

ed è rintracciabile in `<algorithm>`.

La funzione così prototipata restituisce un iteratore che punta al primo

¹³Si vedrà in 3.1.1 come questo meccanismo risulti fondamentale per la gestione del grafico sull'area di disegno.

elemento nel range `[first, last)` per il quale il predicato unario `pred` vale `true`. Se non viene trovato alcun elemento, viene restituito `last`.

remove

La `remove()` nella sua forma più generale funziona come la `find_if()`. `List` però possiede una specializzazione di tale funzione, la quale ha il seguente prototipo:

```
void remove(Predicate pred)
```

La funzione è membro di `list`, diversamente dagli altri algoritmi standard della STL visti in precedenza. Come per `find_if()`, il predicato è unario e scatena la cancellazione quando restituisce `true`, ma la ricerca non si ferma alla prima occorrenza e scorre comunque l'intera lista.

2.4.4 Oggetti funzione

Gli oggetti funzione sono istanze di particolari classi a cui è stato ridefinito `operator()`. Un'oggetto funzione può quindi a tutti gli effetti essere considerato come una funzione, ma rappresenta un meccanismo molto potente di estensione delle funzionalità della libreria.

Per illustrare come gli oggetti funzione lavorino in unione agli algoritmi della libreria standard, vediamo un esempio. Poniamo di avere una classe così definita:

```
class MinMax {
public:
    MinMax();
    int getMax() { return max; };
    int getMin() { return min; };
private:
    int max;
    int min;
};
```

e di possedere una lista `listMinMax` che contiene puntatori a `MinMax`.
Come si può ottenere il massimo dei `max` ed il minimo dei `min`?

La soluzione è la somma dei concetti visti fino a qui riguardanti la STL.
Si definisce un oggetto funzione in questo modo:

```
class CalculateMinMax {
public:
    CalculateMinMax() {
        // inizializzazione opportuna di min e max
    };

    void operator()(MinMax* ref) {
        if ( ref->getMax() > max )
            max = ref->getMax();
        if ( ref->getMin() < min )
            min = ref->getMin();
    }

    int getMax() { return max; };
    int getMin() { return min; };
private:
    int max;
    int min;
};
```

Queste istruzioni:

```
CalculateMinMax limits;
limits = for_each(listMinMax->begin(), listMinMax->end(), limits);
std::cout << limits.getMax() << " " << limits.getMin() << std::endl;
```

stampano a video il risultato cercato¹⁴.

In modo simile a come appena visto, è possibile creare un oggetto funzione che faccia da predicato unario da utilizzare in `find_if()` o

¹⁴Una tecnica analoga viene utilizzata per il calcolo dei limiti di movimento degli elementi di modello sul canvas. L'implementazione viene descritta in 3.1.1.

`remove()`. Questo compito viene semplificato dalla libreria standard che fornisce la struttura `unary_function` utilizzabile come base di derivazione per la definizione di un proprio operatore¹⁵.

¹⁵Si è utilizzato questo meccanismo per la ricerca tramite un ID numerico all'interno di una lista. Si vedrà in 3.2.2 come questa funzionalità è stata implementata.

Capitolo 3

Scelte di progetto

3.1 Gli elementi di modello

Gli elementi di modello¹ sono realizzati sfruttando il meccanismo della derivazione da una classe virtuale pura (a sua volta derivata dalla classe `Gnome::Canvas::Group` per esigenze di programmazione): ciascun elemento implementa l'interfaccia e definisce i propri membri dato. In particolare i membri dato si dividono in due categorie, quelli “grafici” e quelli “dati”: i primi servono per il disegno sul canvas dell'elemento di modello selezionato; i secondi sono necessari per mantenere i valori di alcune variabili ausiliarie. Le due categorie non sono nettamente distinte: una *label* può allo stesso tempo rappresentare un elemento grafico ed un elemento dati. Ma i membri dato che definiscono relazioni tra oggetti grafici sono ovviamente utili solo alla gestione del disegno e non all'aspetto grafico del disegno.

```
class ModelObjBase : public Gnome::Canvas::Group
{
    public:
        ModelObjBase(CanvasMgr*, int);
        virtual ~ModelObjBase();
}
```

¹Si veda anche il paragrafo 1.2.

```
int getId() { return myId; };
CanvasMgr* getManager() { return my_manager_; }

// Gestione del pattern "observer"
virtual void update() = 0;
virtual void notify() = 0;

// Alcuni elementi di modello hanno bisogno di questa funzione, che deve
// andare nella superclasse...
virtual void addChild(ModelObjBase*) {};
virtual void removeChild(int) {};
virtual void killMe() {};

// Gestione del movimento
void moveTo(double, double);

// Definisce i limiti per la gestione dello spostamento
virtual double limitUp() { return 0; };
virtual double limitDown() { return 0; };
virtual double limitRight() { return 0; };
virtual double limitLeft() { return 0; };

// Gestione dei limiti di movimento
virtual double getTopLimit();
virtual double getBottomLimit();
virtual void getLimits(double&, double&);

void virtual setProperties();

void virtual hideHandler() = 0;
void virtual showHandler() = 0;

protected:
    CanvasMgr* my_manager_;
    int myId;
};
```

Listato 3.1: Definizione di ModelObjBase

La classe `ModelObjBase`, definita come nel listato 3.1, è la base di derivazione per gli elementi di modello. Le sue classi derivate sono mostrate in figura 3.1.

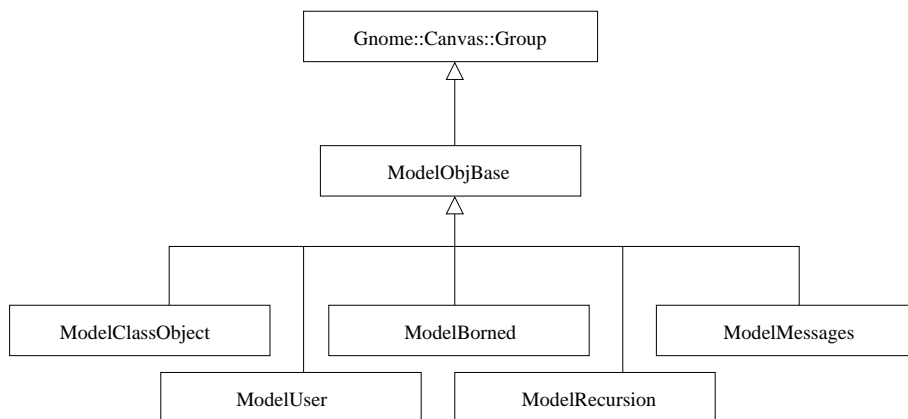


Figura 3.1: Derivazione da `ModelObjBase`

L'utilizzo della derivazione è fondamentale perché permette di gestire in maniera intuitiva il raggruppamento di più elementi grafici primitivi (ad es. rettangoli, linee, testo) per costruire un elemento grafico complesso, che è visto dal canvas come un unico oggetto, cioè come un'istanza di `Gnome::Canvas::Group`. Vedere un elemento complesso come un gruppo permette di gestire in modo semplice il movimento, in quanto è sufficiente spostare il gruppo e non le singole primitive per muovere l'intero modello. Allo stesso tempo però, i modelli devono tenere un riferimento agli altri modelli che dipendono da loro o a cui loro sono collegati. Questi riferimenti sono realizzati tramite liste di puntatori, utilizzate per aggiornare tutti i riferimenti quando avviene un movimento, con un meccanismo legato al *pattern observer* ed analizzato nel paragrafo 3.1.1.

Riassumendo, il diagramma è in pratica un grafo in cui esistono relazioni padre/figlio mantenute da puntatori interni agli elementi di model-

lo, ed utilizzati per calcolare la posizione e stimolare i figli e/o i padri ad aggiornarsi.

3.1.1 Movimento e aggiornamento del grafico

Un problema affrontato nella scrittura del codice è quello relativo al movimento dei modelli, in particolare quando questi sono collegati ad altri. Tale operazione, poiché esistono alcuni vincoli da rispettare, è più complicata del semplice movimento libero. Questi vincoli, sostanzialmente di due tipi, possono dipendere dalla necessità di mantenere la coerenza del disegno o dalle proprietà intrinseche degli elementi di modello.

Per quanto riguarda la coerenza del disegno, i vincoli sul movimento sono necessari affinché sia sempre chiaro chi è il padre di un elemento di modello. Come esempio, si può considerare l'elemento di modello che indica la ricorsione, il quale si sposta solo in direzione verticale, ma con l'accortezza di verificare sempre che le frecce di entrata ed uscita corrispondano ancora alla lifeline del generatore. Semplicemente muovendo verticalmente l'oggetto che rappresenta una ricorsione, la condizione espressa potrebbe non essere sempre verificata. Nella figura 3.2 si vede a sinistra un diagramma coerente ed a destra un possibile problema grafico.

Quando invece consideriamo gli elementi di modello, associamo loro una direzione preferenziale di movimento. Come esempio si può considerare una classe, il cui solo movimento orizzontale è significativo.

Nell'implementazione la gestione del movimento è affidata alla classe `AnchorBox`, la quale gestisce il movimento degli oggetti sul canvas, rispettando eventuali vincoli e definendo una direzione preferenziale. I due problemi sono però risolti in modo diverso: la direzione preferenziale è statica ed è quindi definibile in fase di scrittura del software (nel caso specifico, ogni elemento di modello possiede un `AnchorBox` come membro dato e lo costruisce specificando al costruttore qual è la direzione desiderata); i vincoli

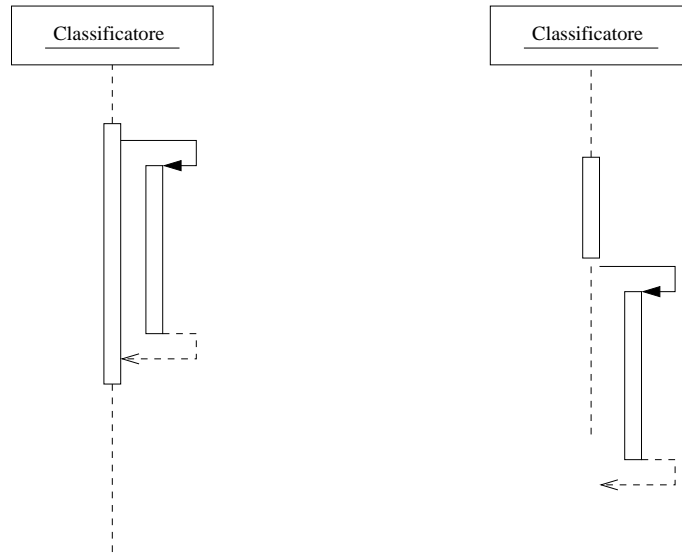


Figura 3.2: Esempio di controllo sui vincoli di movimento

di movimento sono invece dinamici, stabiliti quindi a tempo d'esecuzione e gestiti tramite messaggi tra `AnchorBox` e l'elemento di modello.

Oltre al problema della gestione dei vincoli, esiste anche il problema legato al movimento degli elementi di modello ed al collegamento che esiste a livello grafico tra di essi. Posto che ogni elemento di modello sa quali sono i suoi figli ed i suoi padri (un elemento può essere contemporaneamente entrambi), ogni volta che viene spostato li può aggiornare. In realtà il meccanismo che si usa - derivato dal *pattern observer* - è basato sull'“autoaggiornamento”: ogni elemento di modello è infatti responsabile sia della sua posizione che del suo aspetto.

Il *pattern observer* definisce una dipendenza uno a molti fra oggetti, in modo tale che, se un oggetto cambia il suo stato, tutti gli oggetti dipendenti da questo siano notificati e aggiornati automaticamente. In particolare un oggetto, tutte le volte che viene spostato, informa i suoi figli che qualcosa nel grafico non è più come prima. I figli non sanno cosa sia questo “qualcosa” e non hanno neanche bisogno di saperlo: semplicemente ricalcolano la loro

posizione e si ridisegnano per rispettare la coerenza.

A livello implementativo il tutto è realizzato con due funzioni per ciascun elemento di modello (chiaramente la gestione della coerenza è diversa per ogni elemento): la funzione `notify()` e la funzione `update()`. La prima ha il compito di permettere un dialogo tra padri e i figli, mentre la seconda si occupa di aggiornare l'elemento su cui viene chiamata.

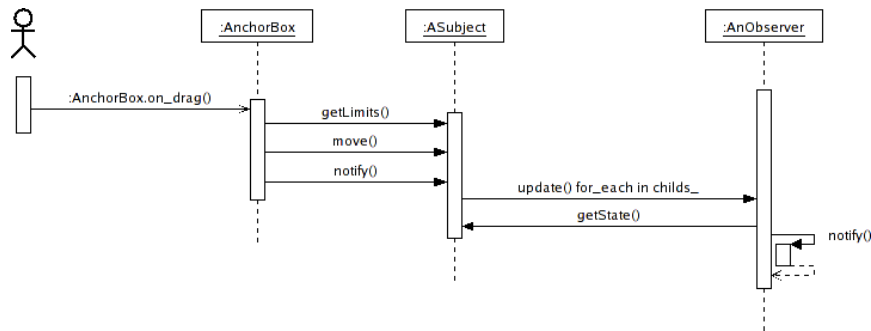


Figura 3.3: Diagramma di sequenza - pattern observer in UMLsdd

Si descrive il funzionamento facendo riferimento alla figura 3.3:

- L'utente trascina l'`AnchorBox` associato all'elemento di modello che vuole spostare;
- `AnchorBox` sposta l'elemento di modello, verificando che il movimento rispetti i vincoli ricevuti dalla `getLimits` e chiama la funzione `notify()` sulla classe che gestisce;
- `notify()` cicla sulla lista dei figli (o degli osservatori) chiamando la funzione `update()`;
- `update()` aggiorna (per ogni osservatore) la posizione e le proprietà grafiche in base alle informazioni ottenute dai padri mediante specifiche funzioni, che possono essere indicate per semplicità col nome di `getState()`.

Questo meccanismo è molto flessibile, poiché sfrutta la derivazione e le funzioni virtuali, ma richiede che ogni elemento di modello mantenga la lista dei propri figli (i padri sono invece mantenuti come membri dato, poiché il loro numero è fisso e conosciuto a tempo di compilazione). Quando `notify()` chiama `update()` su tali elementi di modello, non sa realmente a quale classe appartengano: sarà poi il meccanismo delle funzioni virtuali a scegliere la giusta `update()` in base al tipo dell'oggetto, utilizzando insieme la `for_each()` e la `mem_fun()`, come descritto nei paragrafi 2.4.3 e 2.4.4.

Un'altra cosa molto interessante è il funzionamento di `getLimits()`. Intuitivamente il meccanismo è il seguente: la classe gestita dall'`AnchorBox` attivo cicla sui propri figli facendosi restituire la loro posizione e calcola i limiti di movimento, restituendoli all'`AnchorBox`. L'implementazione è descritta in linea di massima nel paragrafo 2.4.4. La classe che in quell'esempio era `MinMax` adesso è uno degli elementi di modello, mentre l'oggetto funzione `CalculateMinMax` nell'implementazione corrisponde a `GetLimitObjFun`. La funzione `getLimits` ha la seguente dichiarazione:

```
void getLimits(double&, double&);
```

com'è anche possibile vedere nel listato 3.1. Una sua possibile implementazione è:

```
void
ModelRecursion::getLimits(double& top, double& bottom)
{
    GetLimitObjFun childs_limits;
    childs_limits = for_each(childs_-->begin(), childs_-->end(), childs_limits);
    double x1, x2, x3, x4;

    x1 = limitUp() - father_-->limitUp() - SPACE_2;
    x2 = childs_limits.getTopLimit() - limitUp();
    x3 = limitDown() - childs_limits.getBottomLimit();
    x4 = father_-->limitDown() - SPACE_2 - limitDown();
```

```

    double xUpMin, xDownMin;
    xUpMin = x1 > x3 ? x3 : x1;
    xDownMin = x2 > x4 ? x4 : x2;

    top = father_->limitUp() + x1 - xUpMin;
    bottom = father_->limitDown() - getHeight() - x4 + xDownMin;
}

```

È interessante soffermarsi sulle parti più significative. Il ciclo sugli elementi di modello figli viene effettuato dalla `for_each()`. `childs_limits` è un oggetto funzione, che ad ogni passo del ciclo riceve un puntatore a `ModelObjBase` svolgendo queste operazioni:

```

void
GetLimitObjFun::operator()(ModelObjBase* ref)
{
    if ( ref->getTopLimit() < top_limit_ ) {
        top_limit_ = ref->getTopLimit();
    }
    if ( ref->getBottomLimit() > bottom_limit_ ) {
        bottom_limit_ = ref->getBottomLimit();
    }
}

```

I limiti calcolati dall'oggetto `child_limits` possono essere ottenuti tramite le funzioni dell'interfaccia di `GetLimitObjFun`, `getBottomLimit()` e `getTopLimit()`. In realtà la funzione `ModelRecursion::getLimits()` mostrata è più complessa di quanto descritto, poiché nel calcolare i limiti deve tener conto anche dell'esistenza del padre il quale potrebbe imporre un limite più stringente.

Quindi `AnchorBox` per ottenere i limiti di movimento utilizza queste istruzioni:

```

double top, bottom;
owned_->getLimits(top, bottom);

```

trovandosi in `top` e `bottom` i limiti superiori ed inferiori, utilizzabili poi per un confronto nel calcolo dell'offset di spostamento.

Gli oggetti funzione, come si vede in questa implementazione e nell'esempio del paragrafo 2.4.4, rappresentano un metodo molto flessibile per lavorare con i contenitori della libreria standard. Come anche Stroustrup tende a precisare [7], il meccanismo degli oggetti funzione è da preferire alle funzioni globali che mantengono variabili statiche, in quanto l'espansione in linea che il compilatore può effettuare nel primo caso è più efficace.

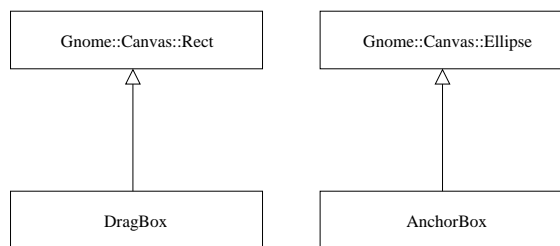


Figura 3.4: Derivazioni delle classi `DragBox` e `AnchorBox`.

3.1.2 Reshaping

La forma degli elementi di modello può essere modificata. Oltre a modifiche “automatiche”, come un rettangolo che si allarga per contenere correttamente il nuovo nome dato ad un oggetto, si devono gestire anche modifiche che l'utente può effettuare sul grafico, come l'allungamento della lifeline di un oggetto.

La classe che gestisce il *reshaping* è la classe `DragBox`. In realtà, poiché ogni modello può essere deformato in un modo che dipende dalla natura stessa del modello, la classe `DragBox` si limita a due sole funzioni: la prima è strettamente grafica e si traduce nel disegno sul canvas di una “maniglia” che l'utente può trascinare per effettuare la deformazione desiderata, in mo-

do molto simile a quanto già visto per `AnchorBox`; la seconda è funzionale e consiste nella chiamata di un segnale (generato dall'evento di "click & drag" del `DragBox` stesso) il cui ricevitore è una funzione in grado di effettuare la deformazione richiesta. Questa funzione è parte dell'interfaccia dell'elemento di modello a cui il `DragBox` appartiene. Quindi `DragBox` è una classe molto povera, che possiede due sole funzioni nella sua interfaccia: una per la rappresentazione grafica² e l'altra per il movimento.

3.2 Canvas Manager

Gli elementi di modello di un diagramma di sequenza UML sono in numero limitato: questo potrebbe permettere una gestione diretta dell'area di disegno da parte della GUI.

In realtà, per permettere una maggiore flessibilità e una possibilità di aggiornamento a nuove versioni dell'UML, si è scelto di gestire l'area di disegno attraverso un *manager*, la classe `CanvasMgr`, che si occupa di istanziare oggetti che rappresentano elementi di modello, previa richiesta da parte dell'utente tramite la GUI dell'applicazione: quando l'utente clicca sul pulsante che rappresenta un elemento di modello, il `CanvasMgr` viene attivato; subito dopo il click dell'utente sull'area di disegno, il `CanvasMgr` istanzia l'oggetto richiesto, lo inserisce sul canvas e in una sua struttura dati³ (realizzata con `list` della STL[7]).

```
class CanvasMgr : public Gnome::Canvas::Canvas
{
```

²Questa funzione è sostanzialmente identica nella forma a quella mostrata nel listato 2.5.

³Questa struttura dati è necessaria per mantenere i riferimenti agli oggetti sul canvas e per poter intervenire in un secondo momento sulle loro proprietà, come quelle di visualizzazione (ad es. posizione e stili grafici).

```
public:
    CanvasMgr(Gtk::Statusbar*);
    virtual ~CanvasMgr();

    int giveMeId();
    int currentId();

    void addToCanvas(ModelObjBase*);
    void removeFromCanvas(int);

    // Questa funzione overloaded definisce i passi da eseguire
    void newStep(KindOfItem); // primo passo
    void newStep(ModelObjBase*); // n passi successivi
    void newStep(); // Chiamata dal canvas... in particolari situazioni
    void deleteModel();
    void deleteAnItem(int);
    void newDraw();

    bool eventCtrl(GdkEvent*); // Deve restituire "false"
    void reset();
    void newProperties();

    void newClassObject() { newStep(CLASS_OBJ); }
    void newClassObjectBornd() { newStep(CLASS_OBJ_BORNED); }
    void newUser() { newStep(USER); }
    void newRecurse() { newStep(RECURSE); }
    void newCalling() { newStep(CALLING); }

    State getState() { return state_; };

    // Salvataggio
    void exportPNG();
    void switchHandlerOfChilds();

protected:
    // Elementi di modello contenuti nel canvas
    std::list<ModelObjBase*>* child_model_obj_;
    Gtk::Statusbar* status_messages_;

    // Conteggio per l'assegnamento di un ID unico ad ogni elemento sul canvas
    int current_id_;
```

```

// Dati membro necessari per l'inserimento di un nuovo oggetto sul canvas
State state_;
KindOfItem new_item_type_;

ModelObjBase* first_ref_;
ModelObjBase* second_ref_;
};

```

Listato 3.2: Definizione della classe `CanvasMgr`

Come si vede dal listato 3.2, la lista che contiene gli elementi di modello è così definita:

```
std::list<ModelObjBase*>* child_model_obj_;
```

ed è quindi un contenitore di puntatori. In questo caso si sfrutta il polimorfismo, poiché è impossibile stabilire a tempo di compilazione il tipo degli elementi di modello che verranno inseriti sul canvas. Questo contenitore è identico a quello che i padri utilizzano per mantenere i figli all'interno degli elementi di modello.

Mentre a tempo di compilazione il `CanvasMgr` non conosce il tipo degli elementi di modello, a tempo di esecuzione questa informazione è nota, permettendo al `CanvasMgr` di muoversi tra stati diversi in un ordine ben preciso in funzione degli stimoli esterni e del tipo del nuovo oggetto: è quindi definibile una macchina a stati, mostrata in figura 3.5, che descrive il comportamento della classe.

Come si vede dalla figura, il canvas manager esegue fundamentalmente tre operazioni:

- Inserzione di un nuovo elemento di modello sul canvas;
- Eliminazione di un elemento di modello dal canvas;
- Modifica delle proprietà di un elemento di modello.

Le operazioni elencate vengono analizzate di seguito, ponendo attenzione ai meccanismi che le governano.

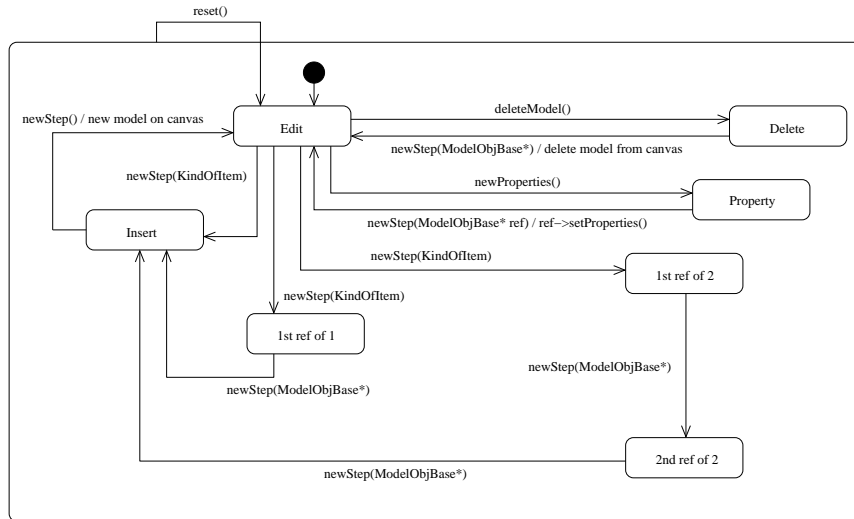


Figura 3.5: Diagramma di stato della classe `CanvasMgr`

3.2.1 Inserimento di un elemento di modello

L'inserimento, insieme all'eliminazione, rappresenta una delle operazioni fondamentali per la gestione del movimento descritta nel paragrafo 3.1.1. È infatti necessario che ogni nuovo elemento si registri ai suoi padri, operazione che può essere effettuata solo in fase di inserimento, poiché una volta assegnati, i riferimenti non vengono più modificati.

Gli elementi di modello sono divisi in tre categorie:

Senza riferimenti → Sono gli elementi di modello che non hanno un padre. In questa categoria rientra il modello “attore” e la “classe/oggetto”.

Singolo riferimento → Sono gli elementi di modello che hanno un solo padre, cioè che mantengono un solo riferimento. In questa categoria troviamo la “ricorsione” e la “classe/oggetto generato”.

Doppio riferimento → Sono gli elementi di modello che hanno due padri. In questa categoria rientra lo “stimolo/messaggio”.

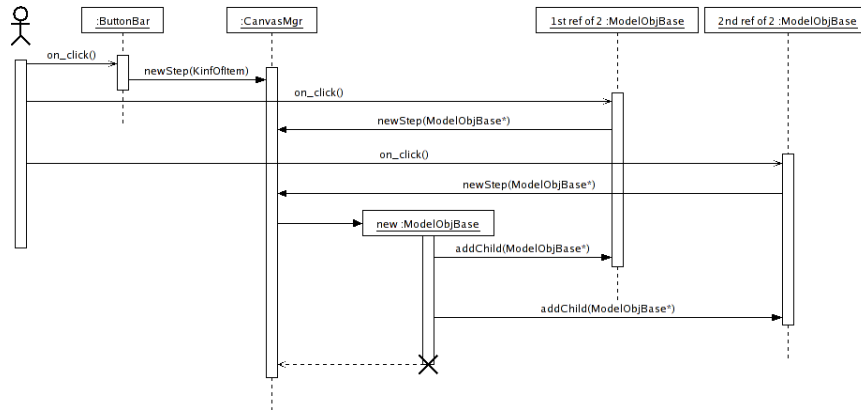


Figura 3.6: Diagramma di sequenza - inserimento di elemento di modello con due padri

Se si guarda il diagramma di stato in figura 3.5, si nota come ci sia un “percorso” per ognuno di questi elementi in uscita dallo stato EDIT. Questo percorso viene scelto implicitamente dall’utente, in base all’elemento di modello che si vuole aggiungere sul canvas.

L’operazione di inserimento viene effettuata in passi. Poniamoci nel caso più complesso di un oggetto con due padri, e in riferimento alla figura 3.6, seguiamo come **CanvasMgr** cambia il suo stato:

- Cliccando il bottone nella barra dei modelli, tramite **ButtonBar** e la chiamata della funzione `newStep(KindOfItem)`, viene attivata la classe **CanvasMgr**, che a questo punto conosce già il tipo del nuovo elemento di modello da inserire. Subito dopo il **CanvasMgr** cambia il suo stato, passando in **1ST REF OF 2**;
- Ogni elemento eleggibile come padre è sensibile al click. Quando questo avviene, il nuovo padre si registra al **CanvasMgr** come primo riferimento attraverso la funzione `newStep(ModelObjBase*)` e muove il **CanvasMgr** verso lo stato **2ND REF OF 2**;

- Al secondo click, la classe `CanvasMgr` ha tutti i riferimenti necessari, e si sposta nello stato `INSERT`;
- Nello stato `INSERT` il `CanvasMgr` istanzia il nuovo elemento di modello, passandogli come parametri sé stesso, un identificatore numerico unico ed i due riferimenti;
- Il costruttore del nuovo oggetto salva i due riferimenti ai padri, calcola la sua posizione iniziale in base ad essi e si registra come figlio attraverso la funzione `addChild()`;
- Il `CanvasMgr` salva il nuovo oggetto nella sua lista.

È necessario notare due cose:

- ogni elemento del canvas ha un ID numerico univoco assegnato da `CanvasMgr`: questo è necessario per le ricerche degli elementi, poiché non ci sarebbe altro modo per indirizzare gli elementi di modello;
- ogni elemento di modello deve essere inserito sia nella lista dei padri, sia in quella del `CanvasMgr`: i primi si occupano della coerenza del grafico, il secondo della definizione delle proprietà e dell'eliminazione.

3.2.2 Eliminazione di un elemento di modello

Così come l'aggiunta di un nuovo elemento sul canvas, anche l'eliminazione pone il problema di lasciare le strutture dati in uno stato consistente. Per ottenere questo risultato si definisce una regola: l'unico ente che può distruggere un elemento di modello è il `CanvasMgr`, mentre gli elementi di modello hanno solo la possibilità di deregistrarne uno dalla propria lista dei figli.

Poiché tutte le strutture dati sono liste, per eliminare un elemento si può utilizzare la funzione specializzata `remove()`, descritta in 2.4.3. La problematica consiste nel rintracciare un elemento dentro la lista e, se necessario,

ottenere un puntatore utilizzabile per chiamare la `delete`. Per raggiungere questo scopo, l'ID numerico unico viene utilizzato insieme ad un oggetto funzione per verificare la proprietà richiesta.

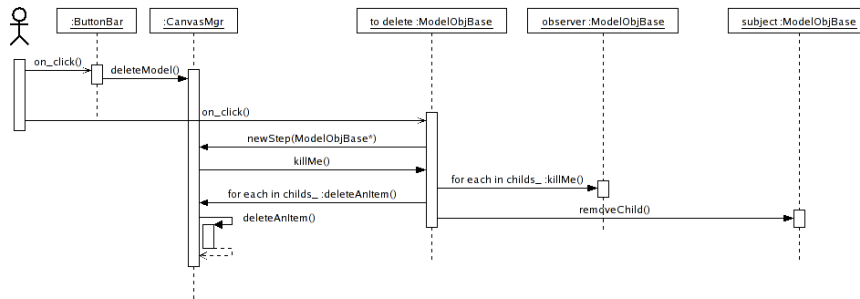


Figura 3.7: Diagramma di sequenza - eliminazione di elemento di modello

Così come per l'inserzione, anche l'eliminazione viene effettuata in alcuni passi:

- L'utente porta la classe `CanvasMgr` verso lo stato `DELETE` attraverso la chiamata di `deleteModel()`;
- Ogni elemento eliminabile è sensibile. Quando un elemento viene cliccato, questo si registra al `CanvasMgr` come vittima;
- Il canvas manager scatena una serie di azioni, prima chiamando `killMe()` sulla vittima, per far in modo che questo si deregistri dai padri ed uccida i proprio figli, poi distruggendo l'elemento tramite la funzione `deleteAnItem()`, cui viene passato l'ID della vittima.

L'eventuale eliminazione a cascata che si può generare è una conseguenza che deriva dalla necessità di mantenere la coerenza nelle strutture dati. Inoltre, poiché il concetto di padre/figlio vale sia a livello software (per la gestione dei puntatori) che a livello grafico, eliminare un elemento che è padre deve necessariamente scatenare anche l'eliminazione dei figli.

La funzione `deleteAnItem()` è così definita:

```
void
CanvasMgr::deleteAnItem(int id)
{
    std::list<ModelObjBase*>::const_iterator iter = find_if(child_model_obj_-->begin()
        , child_model_obj_-->end(), TestEqId(id));
    child_model_obj_-->remove_if(TestEqId(id));

    delete *iter;
}
```

La `find_if()` è necessaria perché restituisce un iteratore il quale punta al primo elemento che soddisfa la condizione di ricerca. La `remove_if` elimina l'elemento dalla struttura dati e successivamente, tramite l'iteratore salvato, viene distrutta la vittima con la `delete`.

```
class TestEqId : public std::unary_function<ModelObjBase*, bool>
{
    public:
        explicit TestEqId(int id);
        bool operator()(ModelObjBase* rif);
        ~TestEqId();

    protected:
        int id_;
};
```

Listato 3.3: Definizione della classe `TestEqId`

```
TestEqId::TestEqId(int id) : id_(id) {}

bool
TestEqId::operator()(ModelObjBase* rif)
{
    return rif->getId() == id_;
}

TestEqId::~~TestEqId()
```

```
{
    // TODO: put destructor code here
}
```

Listato 3.4: Implementazione della classe `TestEqId`

L'oggetto funzione `TestEqId` merita di essere analizzato attraverso i listati 3.3 e 3.4. Nel paragrafo 2.4.4 si è accennato alla possibilità di utilizzare gli oggetti funzione come operatori unari nelle funzioni della libreria standard. In questo caso, dalla definizione della classe, si vede che questa è derivata da una struttura definita nella libreria standard ed utilizzabile come base per la costruzione di un operatore di confronto specializzato. Nel caso in oggetto, `operator()` deve restituire `true` quando l'ID passato al costruttore corrisponde all'ID dell'elemento correntemente valutato.

3.2.3 Cambio delle proprietà di un elemento di modello

Come nei casi precedenti, anche per la gestione delle proprietà viene utilizzato un meccanismo a passi che richiede l'utente come parte attiva: prima clicca il bottone "proprietà" sulla button bar e successivamente seleziona l'elemento di modello di cui vuole modificare le proprietà. Solo in questo momento viene aperta una finestra (istanza di `Gtk::Dialog`) che mostra le vecchie proprietà e permette di definire le nuove.

Diversamente da come sono stati realizzati l'inserimento e l'eliminazione, in questo caso è necessario l'intervento di una terza parte, oltre al `CanvasMgr` e all'elemento di modello per il quale vengono definite le nuove proprietà. Questa terza parte è proprio l'istanza di `Gtk::Dialog`, specializzata attraverso la derivazione per adattarsi ad un particolare elemento di modello. Esistono quindi alcune coppie, elemento di modello e sua finestra delle proprietà, che vengono rese esplicite dal nome delle due classi (ad es. `ModelRecursion` e `ModelRecursionPropertyWin`)⁴.

⁴Questa scelta permette contemporaneamente una più facile gestione dei file e del

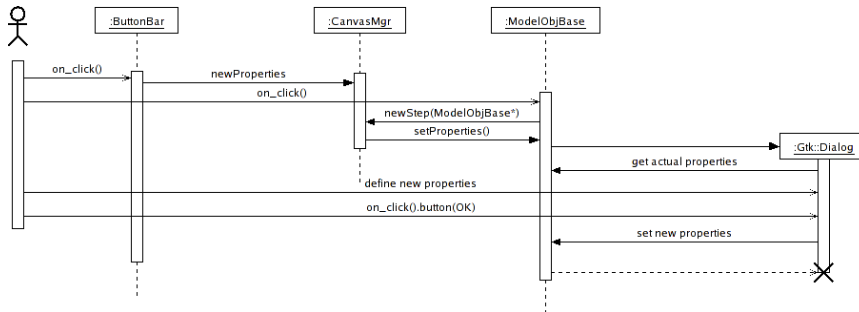


Figura 3.8: Diagramma di sequenza - definizione delle proprietà di un elemento di modello

Come si vede dalla figura 3.8, l'utente è parte attiva non solo nella scelta dell'elemento da modificare ma anche nella scelta delle modifiche da effettuare. Questa modifica viene effettuata quando il `CanvasMgr` si trova nello stato `PROPERTY`, in quanto ne esce solo quando la funzione `setProperties()` termina la sua esecuzione.

```

class ModelClassObjectPropertyWin : public Gtk::Dialog
{
public:
    // Costruttore e distruttore
    ModelClassObjectPropertyWin(ModelClassObject*);
    ~ModelClassObjectPropertyWin();

    // Metodi d'interfaccia
    std::string newLabel();
    bool newIsObject();
    bool newIsConcurrent();

protected:
    // Riferimento al gestito
  
```

codice sorgente. Tale tecnica verrà usata anche nella gestione della GUI, come descritto in 3.3.2.

```

    ModelClassObject* managed_;
    // Widget...
};

```

Listato 3.5: Dichiarazione della classe `ModelClassObjectPropertyWin`

Per analizzare il codice viene preso in considerazione il caso peggiore, la classe `ModelClassObject` ed in particolare la sua `setProperties()`:

```

void
ModelClassObject::setProperties()
{
    ModelClassObjectPropertyWin properties(this);
    int op;

    op = properties.run();
    switch ( op ) {
        case Gtk::RESPONSE_OK: {
            // Aggiornamento delle propriet\`a dell'oggetto
            setLabel(properties.newLabel());
            setIsObject(properties.newIsObject());
            setIsConcurrent(properties.newIsConcurrent());
            // Chiusura della finestra!
            properties.hide();
        } break;
        case Gtk::RESPONSE_CANCEL: {
            properties.hide();
        } break;
        default:
            break;
    }
}

```

La prima istruzione della funzione istanzia un oggetto della classe `ModelClassObjectPropertyWin`, la cui definizione è mostrata nel listato 3.5. Questo oggetto è un widget specializzato per la classe `ModelClassObject`, realizzato per derivazione da `Gtk::Dialog`.

`Gtk::Dialog` è un particolare widget delle GTKmm che disegna una finestra di dialogo. GTKmm lo possiede sia come semplice contenitore, sia specializzato per particolari utilizzi (ad es. selezione dei font, selezione dei colori). Nel progetto è stato utilizzato il contenitore generico, riempito poi con *label*, *check box* e *text box* per permettere la modifica delle proprietà. Ovviamente, poiché ogni elemento di modello possiede proprietà differenti, per ognuno di essi esiste una specializzazione di `Gtk::Dialog`. La specializzazione vista nel listato 3.5 è appunto quella riferita alla classe `ModelClassObject`.

Una caratteristica di questo widget è la sua funzione d'interfaccia `run()`: questa restituisce un intero scelto dal programmatore e diverso per ogni bottone della finestra di dialogo. Esiste quindi un valore di ritorno differente a seconda che l'utente clicchi il bottone "OK" o "Annulla" (o altri bottoni inseriti dal programmatore con compiti specifici). La documentazione delle GTKmm consiglia vivamente di utilizzare come tipo di ritorno valori dell'enumerato `Gtk::ResponseType`, in quanto è poi possibile fare uno switch sui valori dell'enumerato con espressioni letterali ad alto livello: numeri magici in questo contesto potrebbero risultare scarsamente comprensibili. Questo switch è visibile nella definizione in oggetto di `setProperties()`: nel caso `RESPONSE_OK` la funzione legge i nuovi valori e modifica quelli attuali⁵, mentre nel caso `RESPONSE_CANCEL` la funzione si limita semplicemente ad uccidere la finestra di dialogo, senza fare nulla ai valori del modello.

⁵In questo caso non sono state fatte ottimizzazioni, come ad esempio verificare che nuovo e vecchio valore siano diversi ed operare la modifica solo nel caso in cui tale condizione risulti soddisfatta. Probabilmente questo overhead introdotto è scarsamente influente sulla reattività della GUI, che infatti nella pratica non sembra soffrirne.

3.3 Altre scelte

3.3.1 Considerazioni sul pattern Observer

Parlando della coerenza del grafico, specialmente nel paragrafo 3.1.1, si è fatto riferimento al pattern observer, del quale è stata mostrata la struttura generale e l'implementazione specifica nel progetto. La scelta del pattern observer per risolvere questo problema risulta motivata da due aspetti: la necessità di notificare le modifiche subite da un elemento di modello ad altri elementi che da esso dipendono e la capacità di farlo senza conoscere l'identità di chi riceve la notifica. Questi due aspetti, che ad un primo approccio potrebbero risultare ovvii, meritano invece attenzione, in particolare se si considera la riusabilità delle classi che disegnano gli elementi di modello e le possibili estensioni.

Il subject è a conoscenza solo di una serie di observer che rispettano un'interfaccia astratta e non considerano minimamente quale sia la classe concreta che la implementa: ne consegue un accoppiamento astratto e minimale e la possibilità che un nuovo elemento di modello possa essere aggiunto successivamente semplicemente implementando la classe virtuale `ModelObjBase`.

L'accoppiamento viene mantenuto basso anche da altri due fattori:

- La notifica di un cambiamento è di tipo *broadcast*;
- La notifica adotta un modello di tipo *pull*.

Per quanto riguarda il primo fattore, il subject possiede solo una lista degli observer, ma non sa in nessun caso se questi sono interessati alla notifica. Il subject di conseguenza non si interessa nemmeno di quanti siano e del fatto che cambino dinamicamente in numero: la sua unica responsabilità consiste nell'inoltrare la notifica a tutti i presenti.

Per continuare a mantenere basso l'accoppiamento, il subject semplicemente notifica agli observer che c'è un cambiamento e chiede loro di aggiornare

narsi, senza però specificare la causa del cambiamento che si è verificato. Questo tipo di meccanismo è chiamato *pull*. È possibile una soluzione opposta (il modello *push*), che insieme alla notifica allega informazioni aggiuntive sulla causa del cambiamento.

3.3.2 Model-View-Controller (MVC)

Traendo spunto dal concetto di separazione dei compiti introdotto da `SigC++` per la gestione dei segnali (vedi 2.2), nell'organizzazione del codice è stato seguito un criterio analogo di separazione, creando cioè una classe apposita (quasi sempre derivata da una classe appartenente al namespace `Gtk`) relativa alla GUI ed una classe “compagna”, sempre derivata da `SigC::Object`, per la gestione dei segnali generati dall'altra. La scelta descritta rispetta il paradigma Model-View-Controller, che richiede una separazione tra l'informazione e la sua visualizzazione⁶. Tale gestione dei sorgenti di un programma offre la possibilità, nel caso di grandi progetti, di separare nettamente i compiti di programmazione: basterebbe infatti definire in fase di progetto l'interfaccia delle due classi ed affidare la loro implementazione a due programmatori differenti, senza che questi sappiano nulla dell'implementazione della classe “compagna”. In un progetto di dimensioni ridotte e realizzato da una sola persona, come quello descritto nel presente elaborato, la separazione permette di avere a che fare con metodi più piccoli e semplici e su classi più snelle che lavorano in uno spazio ben definito: mantenere nella stessa classe informazioni grafiche e gestori dei segnali avrebbe creato grossi problemi nella risoluzione dei *bug*. La separazione dei file ha inoltre un altro vantaggio, legato alla compilazione del progetto che risulta più

⁶A voler essere rigorosi, in questa gestione si forza un po' il concetto di MVC, perché viene effettuata una separazione tra la visualizzazione da una parte e la gestione della visualizzazione e delle operazioni associate dall'altra.

snella, poiché una eventuale modifica della classe “*controller*” non obbliga alla ricompilazione anche della classe “*view*”.

Per quel che riguarda la nomenclatura, nel progetto viene seguito un criterio standard per assegnare i nomi alle classi: quelle appartenenti alla GUI hanno un nome simbolo, differente a seconda del contesto; le classi associate appartenenti alla zona “*controller*” hanno il nome della classe “compagna” più il suffisso `Controller` (ad es. `MyMenuBar` e `MyMenuBarController`).

Poiché la gestione degli eventi sul canvas richiederebbe che la classe “*Controller*” possa modificare i membri della classe “*View*”, tale separazione non viene fatta. Un metodo per mantenere la separazione consiste nell’uso del costrutto `friend` del C++, ma ciò romperebbe l’*incapsulamento* ed il *data hiding*, il che, rispetto all’accorpamento tra informazione e visualizzazione, sarebbe un danno ben maggiore dal punto di vista del progetto, dato che la separazione tra questi due gruppi non è netta (come già notato in 3.1).

3.4 UMLsdd

UMLsdd (*UML Sequence Diagram Drawing Tool*) è il risultato delle scelte di progetto illustrate in questo capitolo.

In figura 3.9 è possibile vedere uno screenshot dell’applicazione durante la creazione di un diagramma: la finestra di dialogo in primo piano è quella relativa alla modifica delle proprietà di un oggetto generato (che si vede in secondo piano sul canvas).

Il programma implementa i seguenti elementi di modello:

- Classe/Oggetto
- Stimolo/Messaggio
- Oggetto/Classe generato

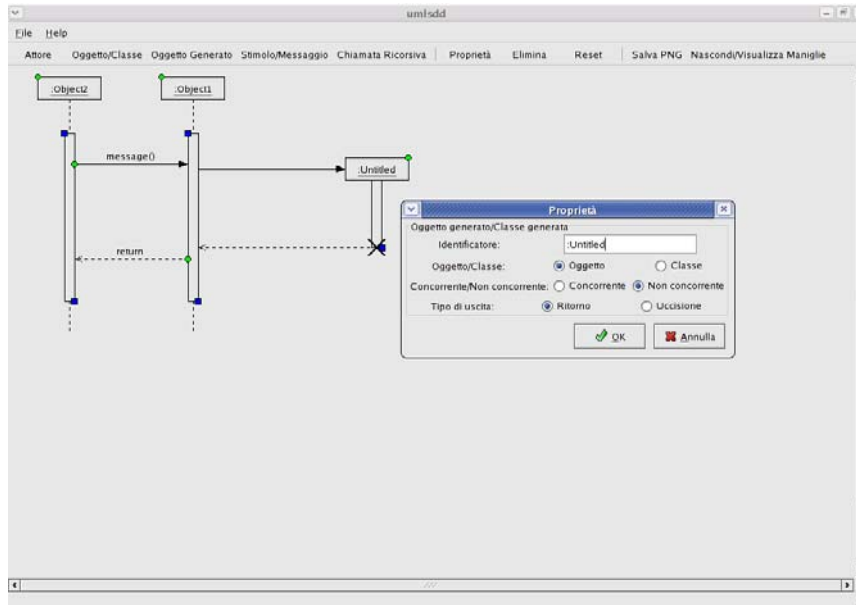


Figura 3.9: Screenshot di UMLsdd

→ Attore

→ Chiamata ricorsiva

Tra gli elementi di modello definiti dallo standard è assente solo lo stimolo/messaggio appositamente per applicazioni real-time (identico a quello standard, ma leggermente obliquo verso il basso) e lo stimolo/messaggio condizionale (quello che lo standard chiama “*branch*”). Il programma, nonostante queste piccole mancanze, nella maggior parte dei casi d’uso non soffre di espressività.

I diagrammi generati possono essere esportati in formato PNG, ma non è possibile effettuare salvataggio e caricamento successivo. Alcuni diagrammi prodotti dal software sono mostrati nelle figure 3.3, 3.6, 3.7 e 3.8.

Conclusioni e ringraziamenti

Realizzare questo progetto è stato molto stimolante. È stato infatti possibile approfondire notevolmente alcune problematiche della programmazione ad oggetti precedentemente affrontate nei corsi di Fondamenti d'Informatica 1, Fondamenti d'Informatica 2 e Ingegneria dei Sistemi Software, prima tramite lo studio di alcuni testi specifici (in particolare [7, 3]), poi con l'applicazione pratica dei concetti studiati alla scrittura del codice.

Il programma è stato scritto nell'arco di 4 mesi, durante le pause dallo studio per la preparazione degli esami. A mio parere il risultato è buono: il programma è funzionale e può essere effettivamente utilizzato per la realizzazione di immagini da inserire in presentazioni o documenti cartacei. Inoltre il codice è stato scritto in un'ottica evolutiva, in modo che risulti agevole aggiungere moduli di ampliamento che rispondano a requisiti assenti nelle mie specifiche, ma che possono rivelarsi ugualmente utili. Si cita, a titolo d'esempio, la gestione del salvataggio e del caricamento e si fa notare a tal proposito che l'OMG definisce un apposito formato XML per il salvataggio dei dati: XMI.

I ringraziamenti sono d'obbligo. Non posso che iniziare dalla mia famiglia: vivo a 1000 Km di distanza e per loro poter “leggere” questo lavoro sarà la conferma che almeno faccio qualcosa!!! Ringrazio poi il prof. Domenici, sempre cortese e disponibile, ed il collega Marcello David: probabilmente senza i suoi consigli starei ancora battendo la testa su

problemi che credevo irrisolvibili. Vorrei inoltre ringraziare i miei colleghi più cari, Susanna e Manuel, che hanno (con)diviso con me, in un modo o in un altro, la mia carriera universitaria sin qui. Ultima, ma non in ordine di importanza, voglio ringraziare la mia amica sincera Giusi, senza il cui stimolo ed il conforto nei momenti difficili non sarei arrivato mai a scrivere queste pagine e senza la cui consulenza queste pagine non avrebbero la forma che hanno (quindi se non vi piacciono, probabilmente è anche un po' colpa sua!). :)

Bibliografia

- [1] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
- [2] OMG Object Management Group. Uml notation guide, 2003.
<http://www.omg.org/cgi-bin/doc?formal/03-03-01>.
- [3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elementi per il riutilizzo di software a oggetti. Prima edizione*. Pearson Education Italia, Milano, 2002.
- [4] Gtkmm home page. <http://www.gtkmm.org>.
- [5] Libsigc++ home page. <http://libsigc.sourceforge.net/>.
- [6] Gtkmm documentation. <http://www.gtkmm.org/gtkmm2/docs/> [Questa documentazione è inclusa con i sorgenti].
- [7] B. Stroustrup. *C++: Linguaggio, libreria standard, principi di programmazione. Terza edizione*. Pearson Education Italia, Milano, 2000.

Elenco delle figure

1.1	Esempi grafici di linee di vita	10
1.2	Esempi grafici di messaggi e stimoli	12
2.1	Parte della derivazione da <code>Gtk::Container</code>	18
2.2	Pattern Decorator	19
2.3	Pattern Iterator	35
3.1	Derivazione da <code>ModelObjBase</code>	43
3.2	Esempio di controllo sui vincoli di movimento	45
3.3	Diagramma di sequenza - pattern observer in UMLsdd	46
3.4	Derivazioni delle classi <code>DragBox</code> e <code>AnchorBox</code>	49
3.5	Diagramma di stato della classe <code>CanvasMgr</code>	53
3.6	Diagramma di sequenza - inserimento di elemento di modello con due padri	54
3.7	Diagramma di sequenza - eliminazione di elemento di modello	56
3.8	Diagramma di sequenza - definizione delle proprietà di un elemento di modello	59
3.9	Screenshot di UMLsdd	65

Elenco dei listati

2.1	Definizione di <code>MainWindow</code>	20
2.2	Costruttore di <code>MainWindow</code>	21
2.3	Utilizzo di <code>SigC++</code> : collegamento con una funzione globale .	26
2.4	Utilizzo di <code>SigC++</code> : collegamento con una funzione membro .	27
2.5	Gestore di segnale default della classe <code>AnchorBox</code>	31
3.1	Definizione di <code>ModelObjBase</code>	41
3.2	Definizione della classe <code>CanvasMgr</code>	50
3.3	Definizione della classe <code>TestEqId</code>	57
3.4	Implementazione della classe <code>TestEqId</code>	57
3.5	Dichiarazione della classe <code>ModelClassObjectPropertyWin</code> . .	59