VISUAL PROLOG PER PRINCIPIANTI

Giovanni Torrero

E-MAIL **giovanni.torrero@email.it**

Testo di riferimento: "Visual Prolog for Tyros" di Eduardo Costa. [**tyros**]

Indice

Capitolo 1. Introduzione 5
- 1.1. Creare un progetto con V.P.7.1 5
- 1.2. Creare forms 8
- 1.3. Eventi 14
- 1.4. P.I.E. - Program Inference Engine 18

Capitolo 2. Il Prolog 21
- 2.1. Goal 21
- 2.2. Errori logici 27
- 2.3. Il CUT 29
- 2.4. Le Liste 33
- 2.5. La Ricorsione 34

Capitolo 3. Le classi 37
- 3.1. Il Programma calcolo_fattoriale 37
- 3.2. Il programma mappaBaseDati 61
- 3.3. Il programma SoluzioniMultiple 80
- 3.4. Le applicazioni Console 93
- 3.5. Menù - toolbar 96

Indice analitico 109

Bibliografia 111

Bibliografia 113

CAPITOLO 1

Introduzione

1.1. Creare un progetto con V.P.7.1

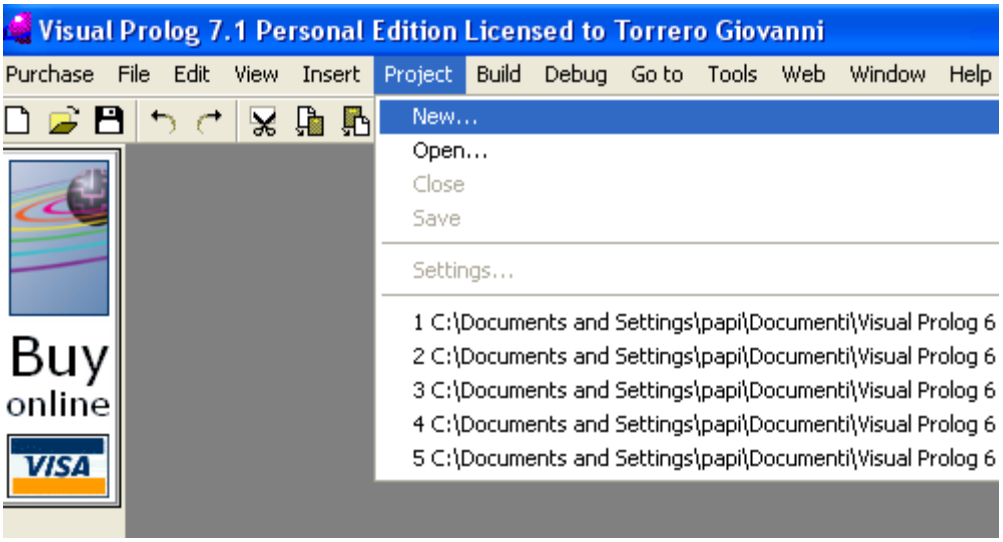

FIGURA 1.1.1. Project-New

Lanciando V.P.7.1 e scegliendo Project-New come in figura (1.1.1) di pagina 5 si inizia un nuovo progetto. Successivamente a tale scelta comparirà la finestra di dialogo rappresentata in figura (1.1.2) di pagina 6:

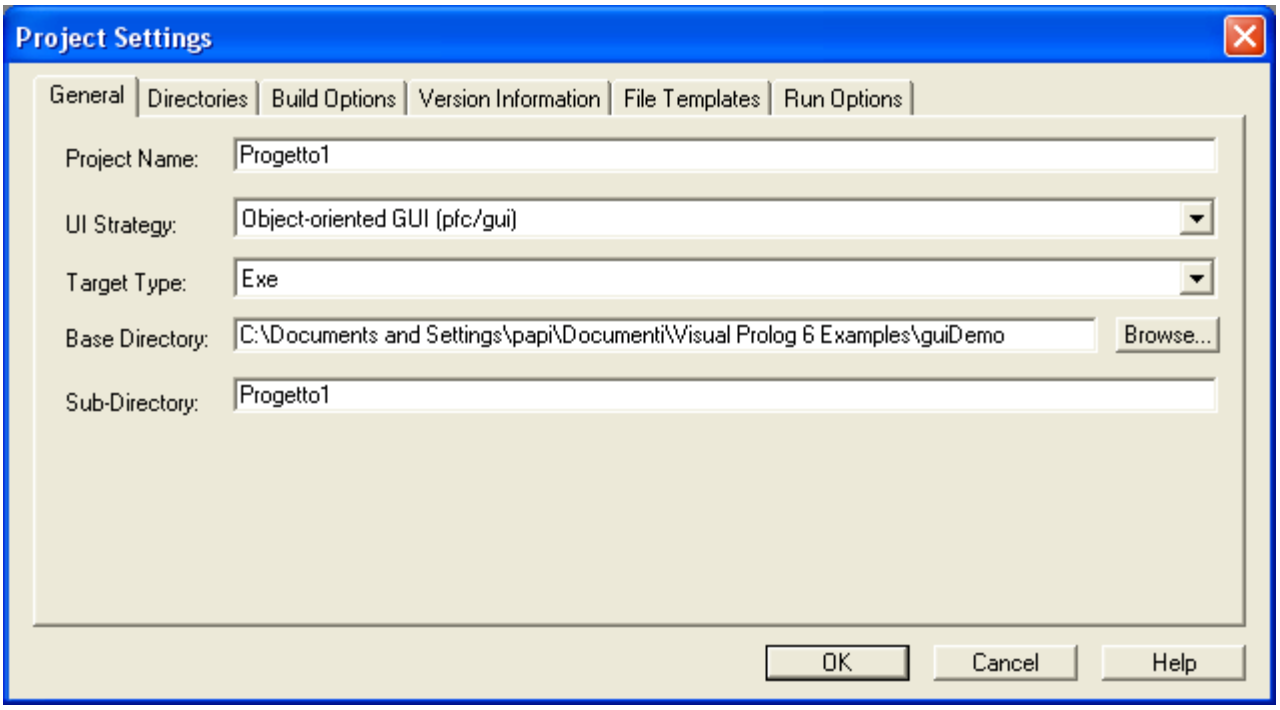

FIGURA 1.1.2. Project Settings

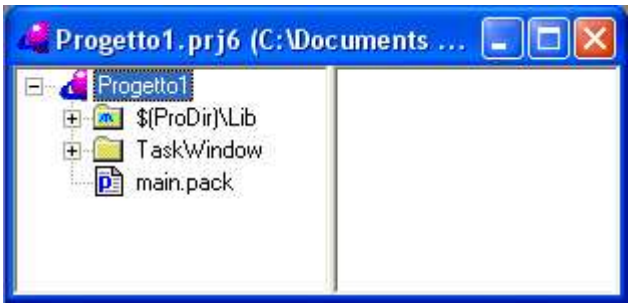

FIGURA 1.1.3. Albero del progetto

É sufficiente riempire la scheda General lasciando le altre schede invariate, scegliendo Ok si ottiene la finestra di dialogo di figura (1.1.3) di pagina 6. In questa figura si vede la struttura ad albero del progetto, il quale è già una applicazione completa anche se non fa nulla, la si può compilare ed eseguire. Per compilare una applicazione si fa la scelta indicata nella figura (1.1.4) di pagina 7, il progetto viene compilato e il risultato è il file eseguibile *Progetto1.exe* che nel nostro caso si trova nella directory

```
C:\Documents and Settings\papi\Documenti\Visual Prolog 6 Examples\guiDemo\Progetto1
```

La scelta di questa directory la si fa inizialmente con la scheda di figura (1.1.2) di pagina 6 quando si sceglie la directory del progetto stesso.

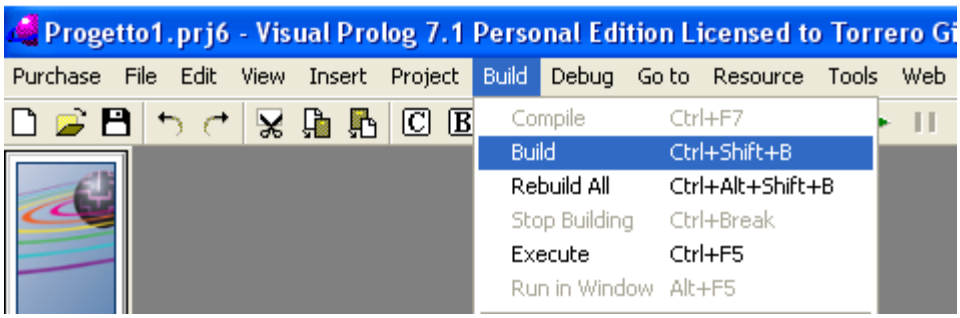

FIGURA 1.1.4. Build

Andando nella directory anzidetta e facendo doppio click su Progetto1.exe viene eseguita la nostra applicazione la quale produce la finestra riportata nella figura 1.1.5 di pagina 7

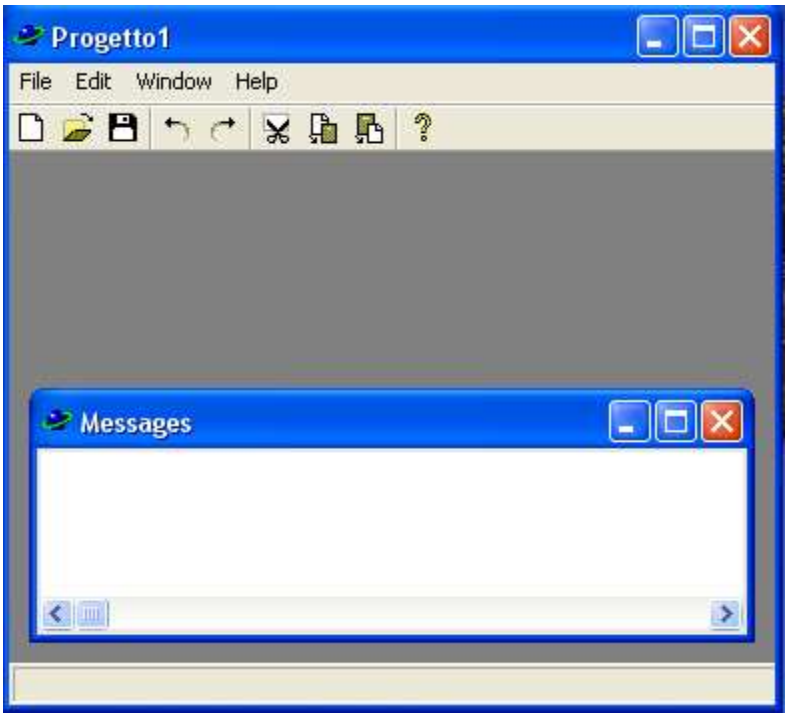

FIGURA 1.1.5. Esecuzione del programma

L'applicazione non fa nulla, vediamo come è possibile darle un contenuto.

1.2. Creare forms

Una form è una finestra alla quale si possono aggiungere dei controlli. Per creare una nuova form bisogna seguire i passi indicati nella figura 1.2.1 di pagina 8

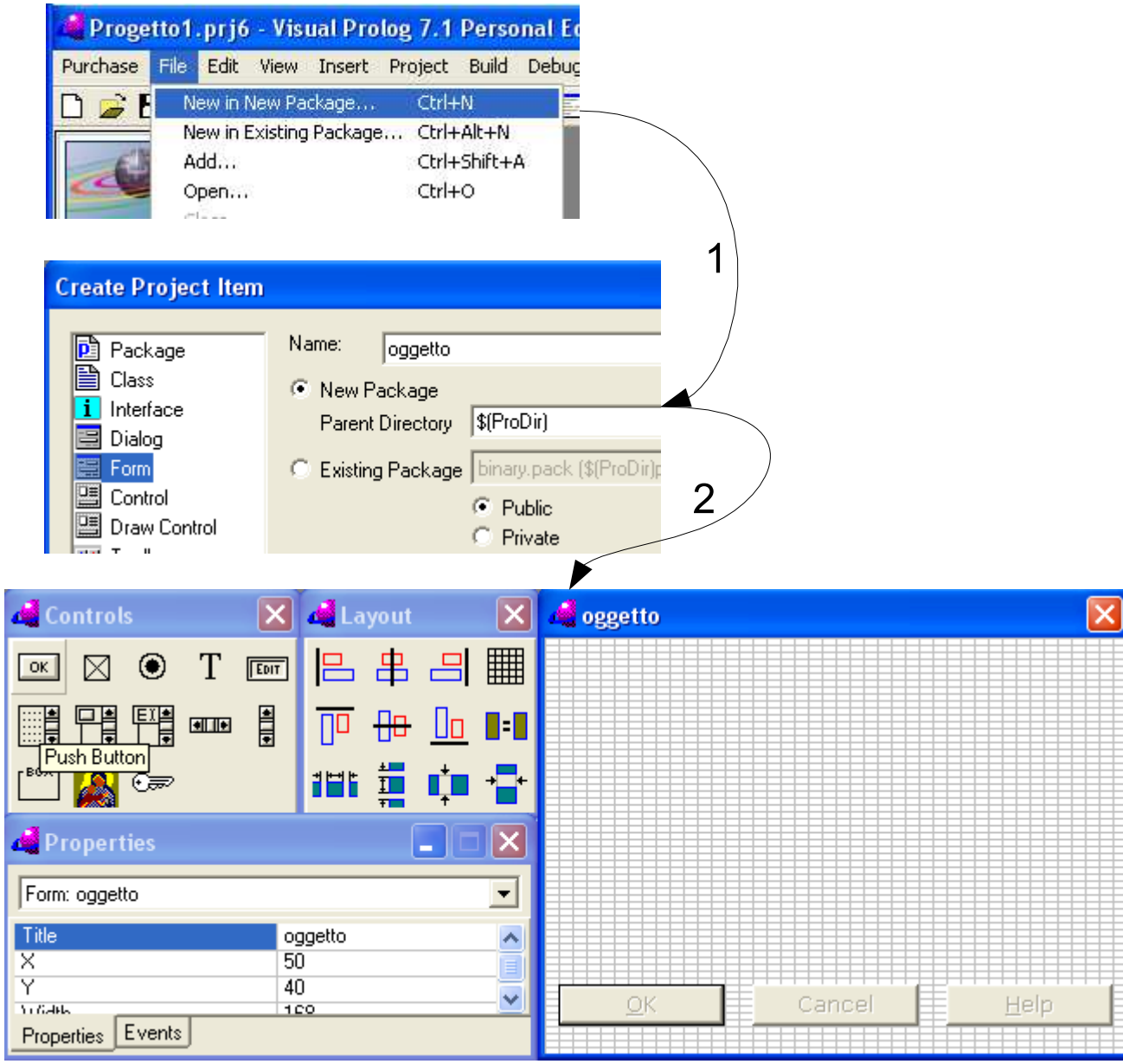

FIGURA 1.2.1. Creare form

Nella finestra di dialogo Create Project Item , dopo aver riempito la casella Name scegliere il pulsante Create .

Se, dopo aver fatto quanto sopra, provate ad eseguire il programma vi accorgerete che non è possibile visualizzare la form appena creata. Bisogna collegare l'esecuzione della form ad uno degli eventi del mouse. Nel menù File dell'applicazione vorremmo che la voce New fosse collegata all'apertura di questa form. Andiamo sulla finestra Progetto1.prj6 dove si trova il diagramma ad albero del progetto e facciamo le scelte indicate nella figura (1.2.2) di pagina 9 , comparirà la finestra di dialogo di figura (1.2.3) pagina 9.

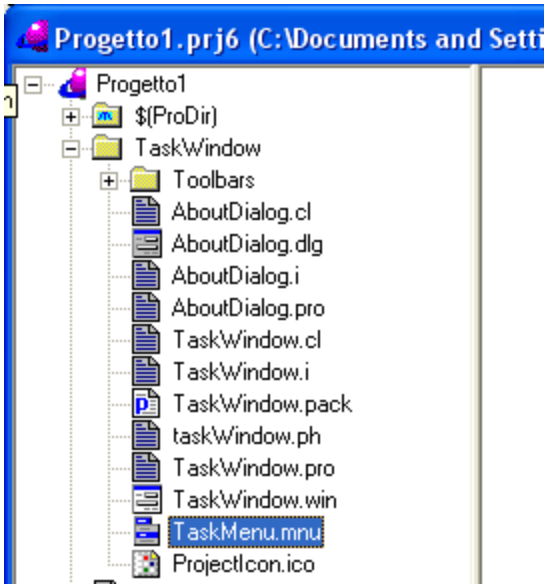

FIGURA 1.2.2. Aprire un form

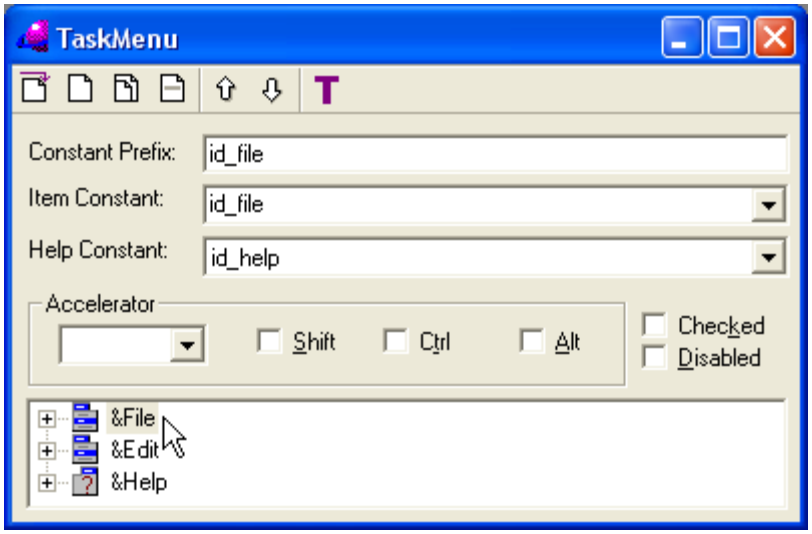

FIGURA 1.2.3. Aprire un form

Scegliamo &File - &New , la finestra TaskMenu diventerà quella riportata in figura (1.2.4) di pagina 10. Come si vede la casella Disabled è selezionata, fare click sul segno di spunta per annullare la selezione .

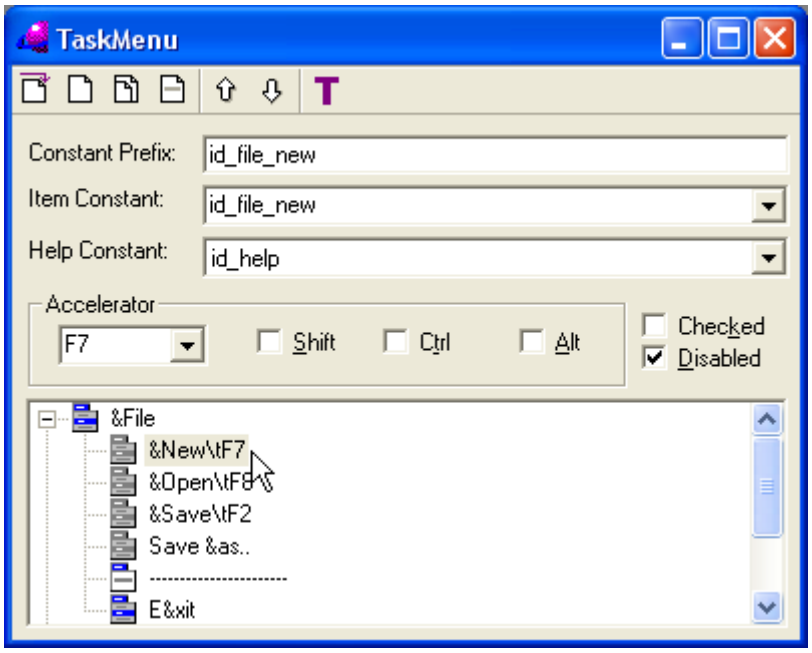

FIGURA 1.2.4. Aprire un form

Dopo aver fatto quanto detto sopra ritorniamo sul diagramma ad albero del progetto come in figura (1.2.5) di pagina 11 e facciamo click con il tasto destro del mouse su TaskWindow.win compare una finestra pop-up come in figura (1.2.6) di pagina 11 .

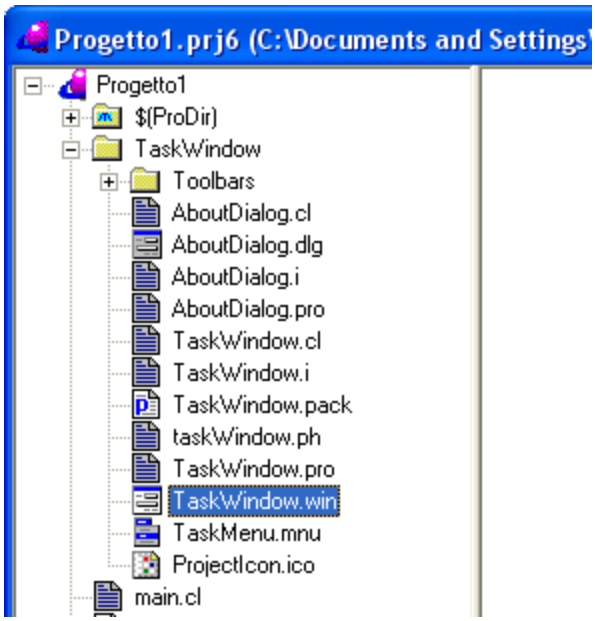

FIGURA 1.2.5. Task menu win

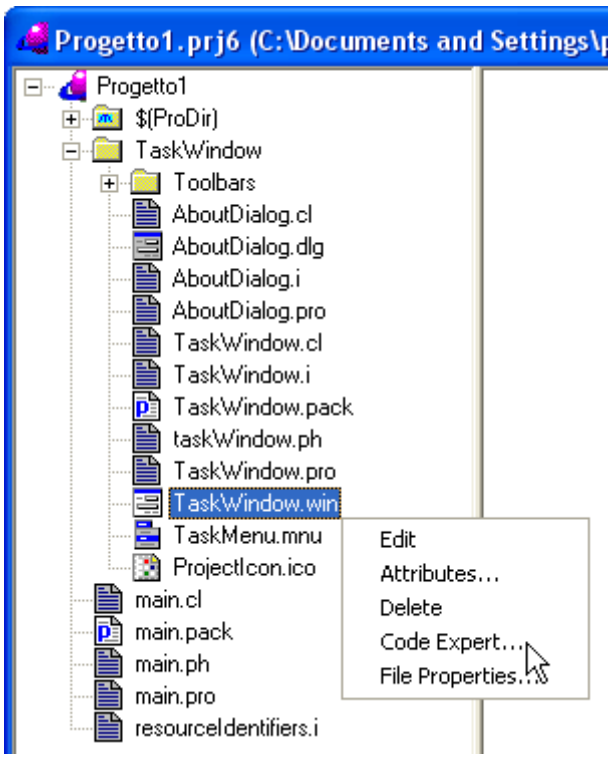

FIGURA 1.2.6. Code Expert

Scegliamo la voce CodeExpert ed apriremo la finestra di dialogo riportata in figura (1.2.7) di pagina 12 , scegliamo Id_file_new e facciamo click sul pulsante Add posto in basso a destra nella finestra, la finestra di dialogo diventerà quella riportata in figura (1.2.8) di pagina 13 .

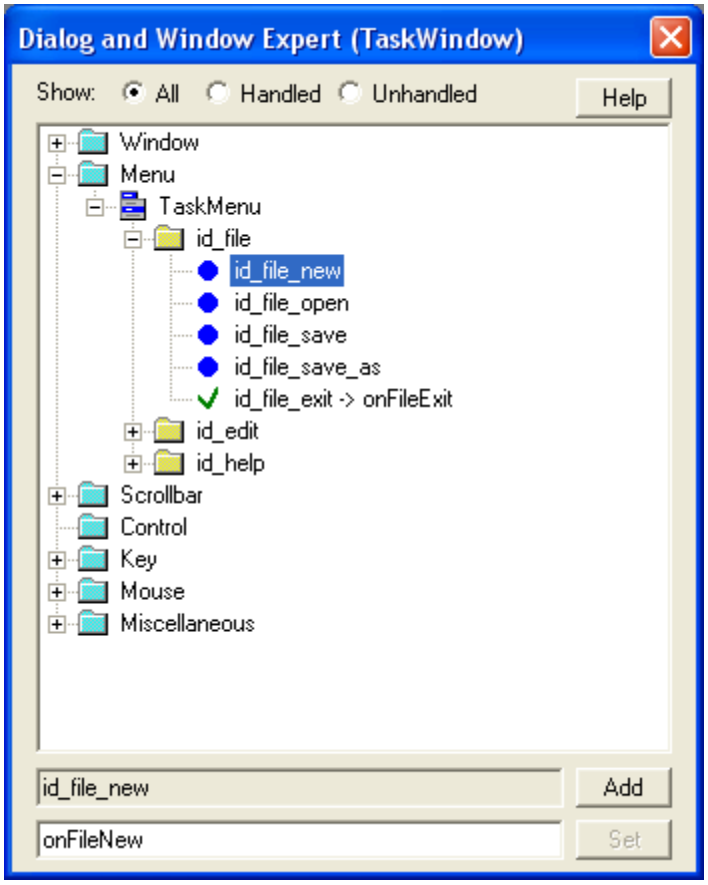

FIGURA 1.2.7. Code expert

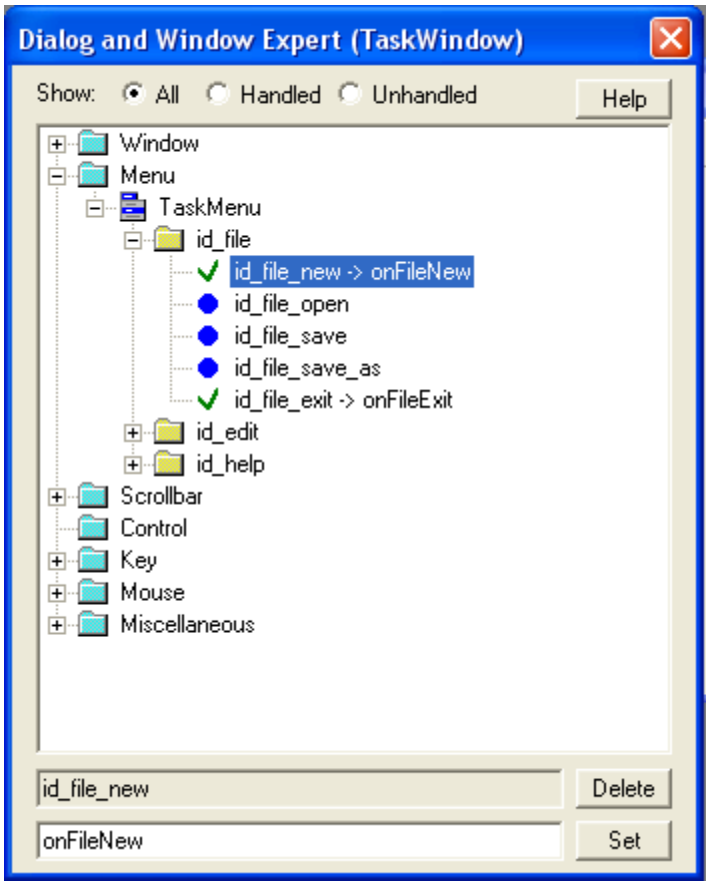

FIGURA 1.2.8. Code Expert

Fate doppio click sulla parte evidenziata della figura (1.2.8) di pagina 13 e verrà visualizzata la finestra TaskWindow.pro che contiene il codice Prolog relativo alla gestione dell'evento considerato (figura (1.2.9) di pagina 14).

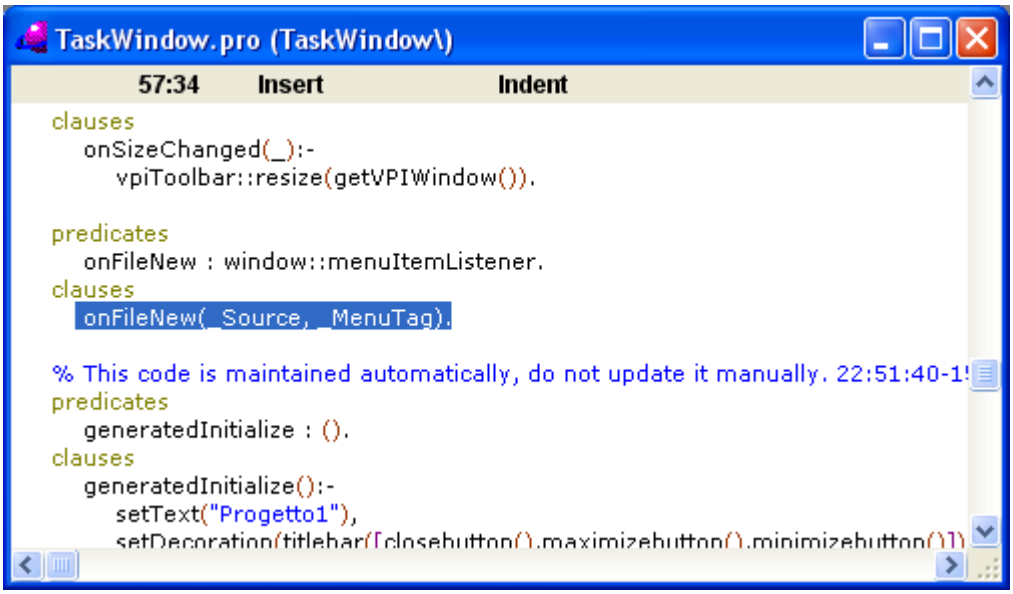

FIGURA 1.2.9. Frammento di codice

Cancellate il codice evidenziato e sostituitelo con questo:

```
onFileNew(W, _MenuTag) :- X= oggetto::new(W), X:show().
```

Compilate ed eseguite, noterete che facendo click su File - New compare la finestra della form oggetto.

1.3. Eventi

Sulla barra del menù scegliete View → Project Window comparirà una finestra dentro la quale scegliete Oggetto.frm ed arriverete a quanto illustrato nella figura (1.3.1) di pagina 15

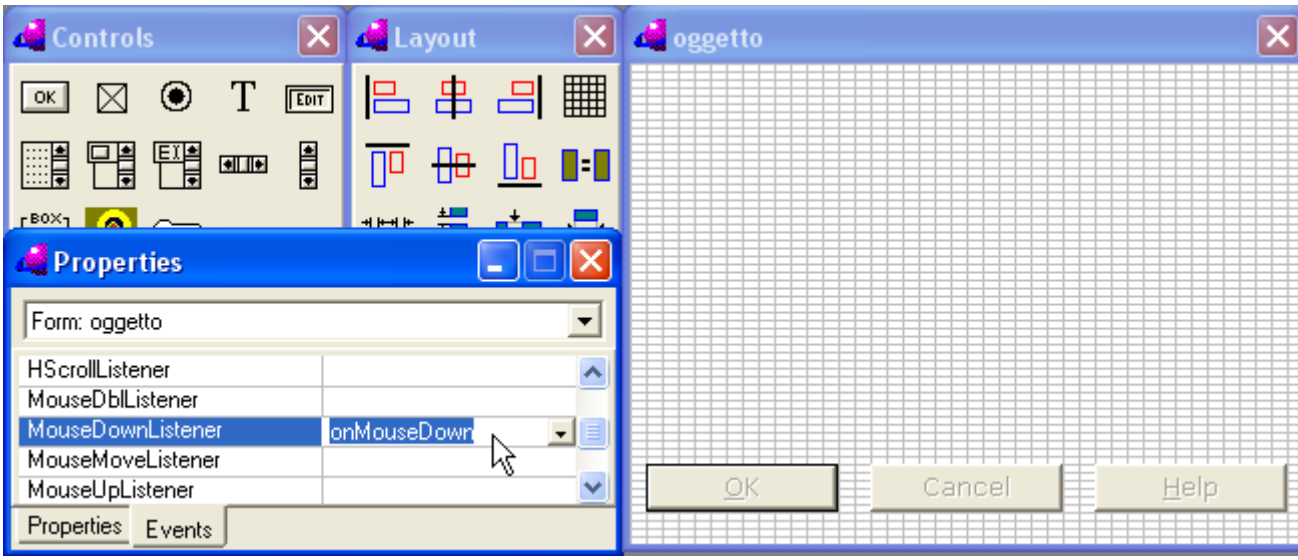

FIGURA 1.3.1. Progettare un form

Nella finestra Properties scegliete: Event → MouseDownListener → onMouseDown, fate doppio click su MouseDownListener ed apparirà la finestra (1.3.2) di pagina 15.

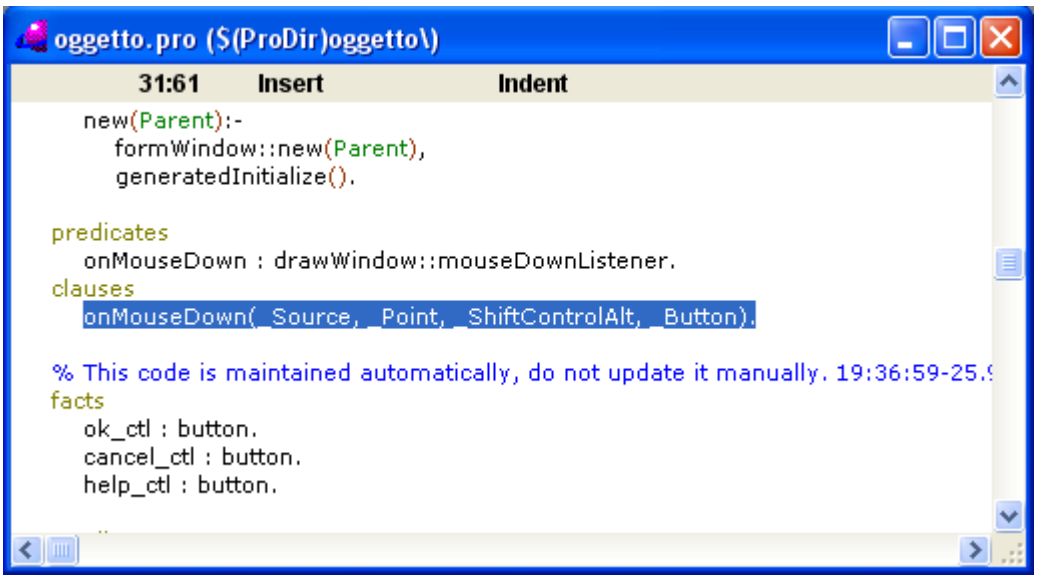

FIGURA 1.3.2. Evento del mouse

Sostituite la parte di codice evidenziato con il seguente:

```
clauses
```

```
onMouseDown(S, Point, _ShiftControlAlt, _Button) :-
W= S:getVPIWindow(), Point= pnt(X, Y), vpi::drawText(W, X, Y, "Hello, Worl
```

Compilate ed eseguite, ogni volta che fate click nella finestra Oggetto comparirà la scritta *Hello, world!* .

Il metodo scelto per scrivere *Hello world* non è molto ortodosso. Se fate click nella form in punti vicini tra loro vi accorgerete che le scritte si sovrappongono. Una procedura migliore consiste nel cancellare una regione rettangolare attorno al punto nel quale si è fatto click e scrivere in detta regione *Hello, world*. L'ascoltatore MouseDownListener, della form oggetto, è in grado di rilevare l'abbassamento del tasto destro del mouse e di lanciare la procedura OnMouseDown. Rifate il progetto precedente portandolo fino al punto in cui dovete riscrivere la procedura OnMouseDown a pagina 16 a questo punto la procedura OnMouseDown, in oggetto.pro, sostituitela con questa:

```
class facts
   mousePoint:pnt := pnt(-1, -1).
predicates
   onMouseDown : drawWindow::mouseDownListener.
clauses
   onMouseDown(_S, Point, _ShiftControlAlt, _Button) :-
         mousePoint := Point,
         Point= pnt(X, Y),
         R= rct(X-8, Y-8, X+60, Y+8),
         invalidate(R).
```

Il codice precedente non è ancora del tutto comprensibile, ma si intuisce che si parla di un rettangolo legato al punto in cui è stato fatto click con il mouse e che questo rettangolo viene invalidato. Invalidare un rettangolo significa cancellarlo ed emettere un evento del tipo onPaint, questo evento ha l'ascoltatore PaintResponder nella form Oggetto. Ricevuto l'evento l'ascoltatore attiva la procedura onPaint la quale ridisegna il rettangolo . Quindi andate sul menù e scegliete

View → Project Window nella finestra di dialogo che compare scegliete oggetto → oggetto.frm, compariranno le quattro finestre di dialogo relative alla form scelta, nella finestra Properties scegliete la scheda Events e dentro questa scheda andate a cercare PaintResponder ed attivate la corrispondente procedura onPaint, fate doppio click come indicato nella figura (1.3.3) di pagina 17

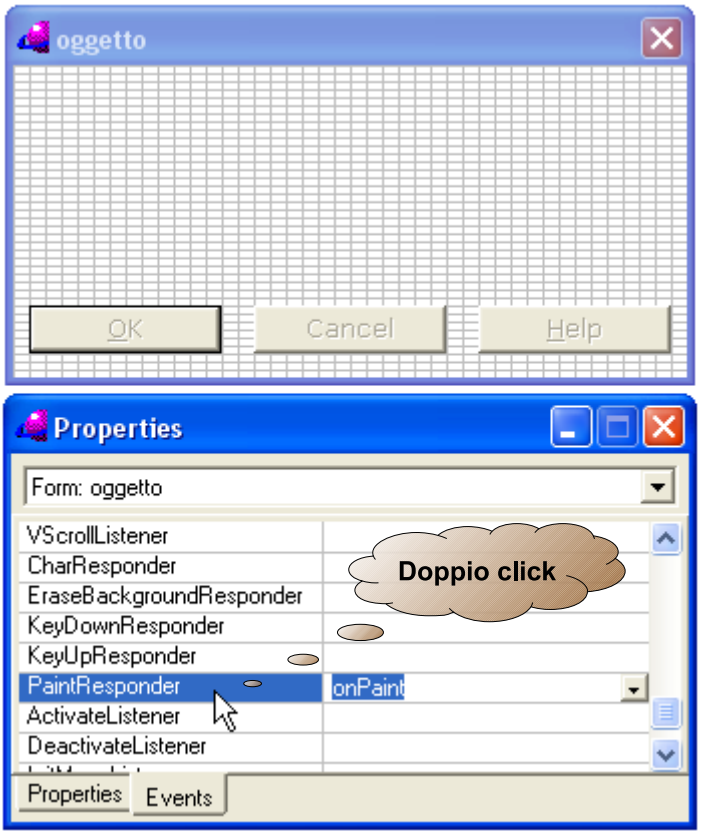

FIGURA 1.3.3. Paint

Otteniamo il seguente frammento di codice nella finestra oggetto.pro

```
predicates
     onPaint : drawWindow::paintResponder.
clauses
     onPaint(_Source, _Rectangle, _GDIObject).
```

Questa parte va modificata nel modo seguente:

```
predicates
     onPaint : drawWindow::paintResponder.
clauses
    onPaint(_Source, _Rectangle, GDIObject) :-
      if pnt(X, Y)= mousePoint, X>0
      then
        mousePoint := pnt(-1, -1),
        GDIObject:drawText(pnt(X, Y), "Hello, World!", -1)
      else succeed() end if.
```

Compilate ed eseguite, quando fate click nella finestra oggetto comparirà la scritta *Hello, world* senza sovrapporsi alle altre.

1.4. P.I.E. - Program Inference Engine

Andate al seguente URL: `http://www.visual-prolog.com/` nella sezione NEW AND UPDATED EXAMPLES troverete un link che vi consente di scaricare il file *pie.zip*. Estraete i file in una directory, lanciate Visual Prolog 7.1, caricate il progetto *PIE.prj6*, compilate e nella directory *Exe* del progetto troverete l'applicazione *PIE.exe*, lanciatela ed otterrete la finestra di figura 1.4.1 di pagina 18

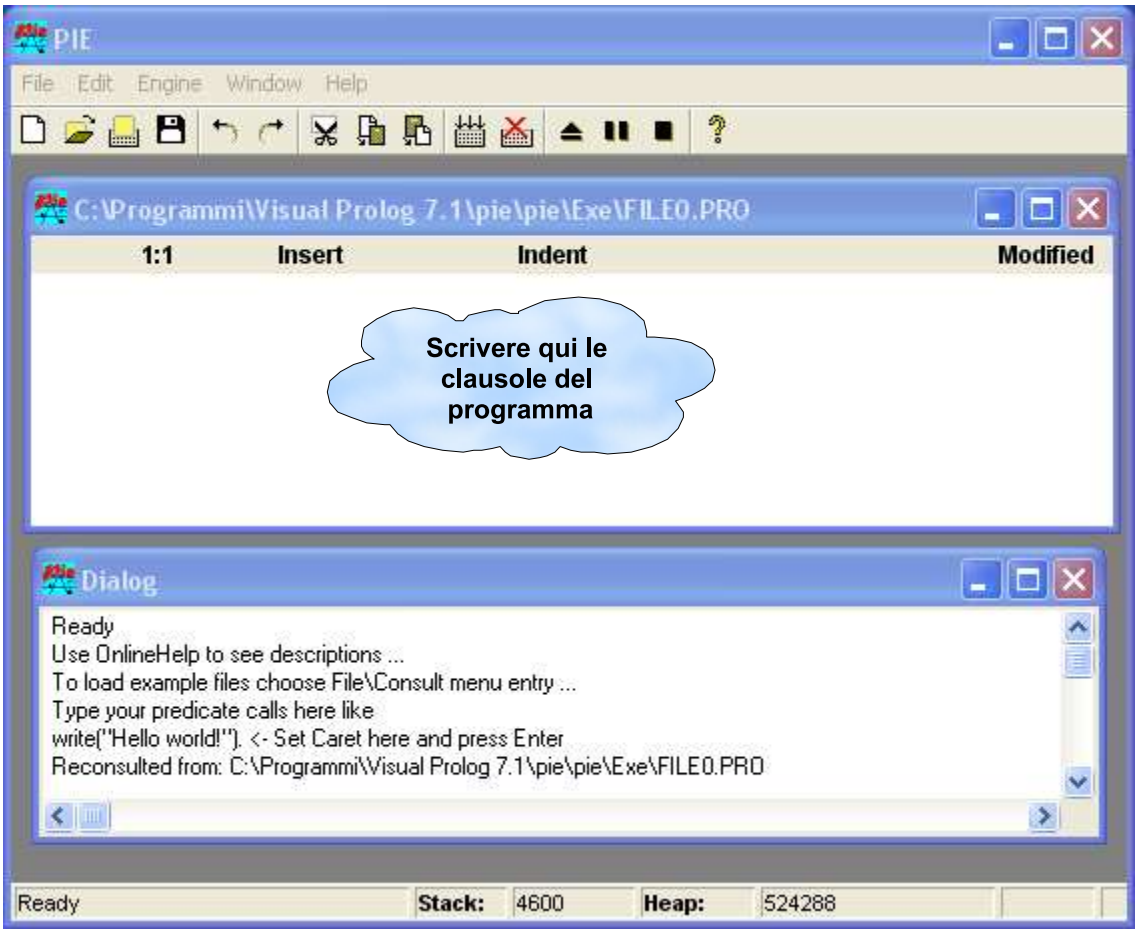

FIGURA 1.4.1. L'applicazione PIE.exe

Nell'apposita finestra indicata nella figura scriviamo le seguenti clausole

```
father("Bill","John").
father("Pam","Bill"):

grandfather(Person,Grandfather) :-
   father(Person,Father),
   father(Father,Person).
```

Queste clausole vanno lette nel seguente modo:

- John è il padre di Bill
- Bill è il padre di Pam
- GrandFather è il nonno di Person se
  - Father è padre di Person, e
  - GrandFather è il padre di Father.

Le prime due clausole vengono anche chiamate *fatti*, mentre la terza viene chiamata *regola* . Le regole sono caratterizzate dal fatto che nel loro corpo è presente il simbolo :- che sta la posto della parola *se*. I fatti, dal punto di vista matematico, esprimono una relazione, nel nostro caso *"essere padre di"*, queste relazioni vengono anche chiamate *predicati*, le regole ci dicono che un fatto è vero se sono contemporaneamente veri alcuni altri fatti. Una regola è composta da due parti, come è illustrato nella figura 1.4.2 di pagina 19, dalla testa, la parte che precede il simbolo :-, e dal corpo, la parte che segue lo stesso simbolo.

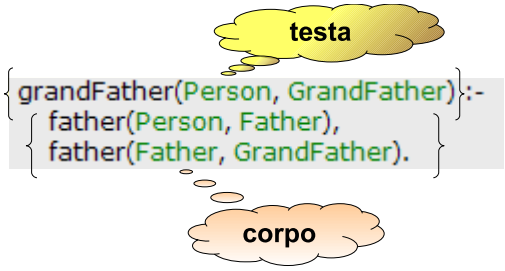

FIGURA 1.4.2. Regole

I fatti possono considerarsi regole prive di corpo. Il fatto che costituisce la testa è vero se sono veri tutti i fatti che formano il corpo. L'insieme di tutte le regole forma una *teoria* oppure una *base di conoscenze* . Osserviamo che gli argomenti dei fatti sono scritti in modo diverso, in *father("Bill","John")* sono racchiusi tra apici, mentre in

```
1 grandFather(Person, GrandFather) :-
2 father(Person, Father),
3 father(Father, GrandFather).
```

gli argomenti non sono racchiusi tra apici ed iniziano tutti con una lettera maiuscola. Gli argomenti che sono racchiusi tra apici, oppure che iniziano con una lettera minuscola sono argomenti legati , "BILL" non è una persona indeterminata, è un indicatore di qualche cosa di fisso di determinato, mentre invece gli argomenti che non sono racchiusi tra apici e iniziano con una lettera maiuscola non sono legati, non rappresentano qualche cosa di fisso, ma il loro valore può variare, a stabilirlo è il meccanismo di ricerca del Prolog, per capirlo dobbiamo introdurre il concetto di goal.

Andiamo nella finestra di figura (1.4.1) di pagina 18, nella quale abbiamo già scritto la teoria sopra riportata, facciamo click sulla icona Reconsult come indicato nella figura 1.4.3 di pagina 20 portiamoci nella finestra dialog e scriviamo il seguente goal:

```
father("Bill","John")
```

premiamo Ok , nella finestra dialog comparirà quanto riportato nella figura 1.4.3 di pagina 20

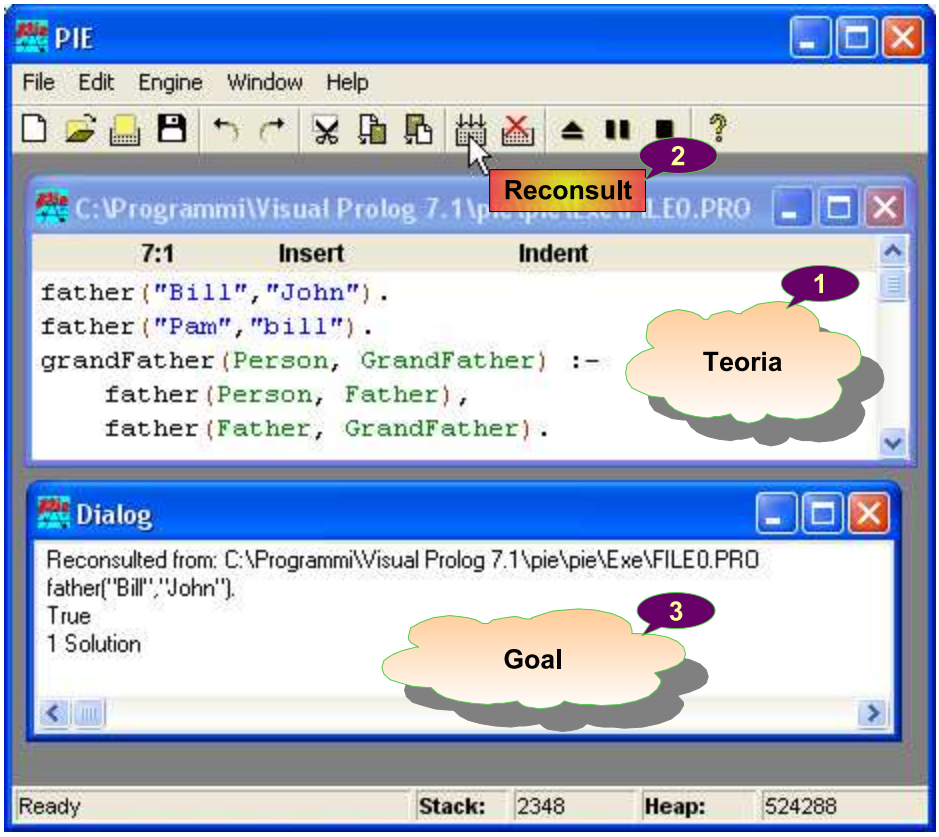

FIGURA 1.4.3. Esempio

CAPITOLO 2

Il Prolog

2.1. Goal

Mediante l'uso di PIE cerchiamo di spiegare come il Prolog cerchi di soddisfare i goal che gli vengono proposti. A tale scopo utilizziamo una teoria più elaborata:

```
punto("A").
punto("B").
punto("C").
punto("D").
punto("E").

retta(a).
retta(b).
retta(c).
retta(d).

appartiene(a,"B").
appartiene(c,"B").
appartiene(a,"E").
appartiene(d,"D").
appartiene(b,"E").
appartiene(a,"C").
appartiene(b,"D").

incidenti(X,Y) :-
  retta(X), retta(Y),
  appartiene(X,P),
  appartiene(Y,P).

parallele(X,Y) :-
  retta(X), retta(Y),
  not(incidenti(X,Y)).

allineati(X,Y,Z) :-
  punto(X), punto(Y), punto(Z),
  retta(A),
  appartiene(A,X),
```

```
33    appartiene(A,Y),
34    appartiene(A,Z).
```

Riportiamo questo listato su PIE e, nella finestra Dialog scriviamo il seguente goal:

appartiene(a,X).

ed otteniamo i risultati riportati in figura 2.1.1

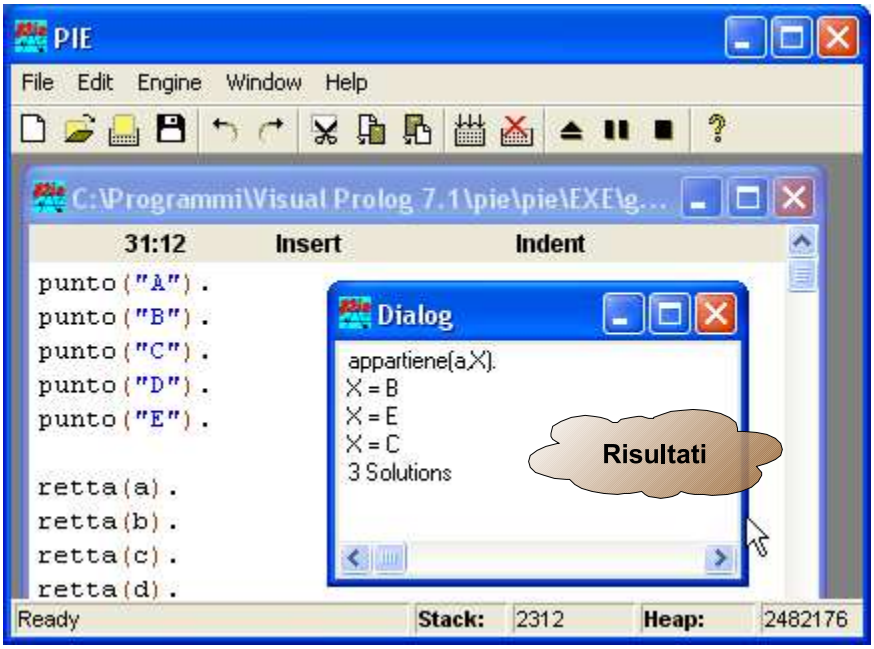

FIGURA 2.1.1.

Quando viene lanciato un goal il Prolog parte dall'inizio della base delle conoscenze e, muovendosi dall'alto verso il basso, cerca un predicato che coincida con il predicato della testa del goal. Nel nostro caso abbiamo lanciato il goal *appartiene(a,X).*, il predicato da cercare è *appartiene* , partendo dall'alto della teoria e muovendosi verso il basso il primo predicato di quel tipo che si incontra è quello indicato in rosso:

punto("A").

punto("B").

punto("C").

punto("D").

punto("E").

retta(a).

retta(b).

retta(c).

retta(d).

→appartiene(a,"B").

appartiene(c,"B").

appartiene(a,"E").

appartiene(d,"D").

appartiene(b,"E").

appartiene(a,"C").

appartiene(b,"D").

Il predicato *appartiene(a,"B").* differisce dal goal *appartiene(a,X).* soltanto per la variabile X la quale, essendo variabile, può benissimo legarsi al valore costante "B", legandosi in questo modo il goal resta perfettamente sovrapponibile alla clausola[1] della teoria. Il goal resta così soddisfatto e il Prolog ne dà notizia avvertendo che X si è legata a "B". La teoria potrebbe essere in grado di soddisfare il goal anche in altri modi, allora la variabile X viene slegata dal valore "B" e la ricerca riprende dall'alto in basso partendo dalla clausola successiva a quella già considerata. La prima clausola che si incontra è quella riportata in rosso:

punto("A").

punto("B").

punto("C").

punto("D").

punto("E").

retta(a).

retta(b).

retta(c).

retta(d).

appartiene(a,"B").

→appartiene(c,"B").

appartiene(a,"E").

appartiene(d,"D").

appartiene(b,"E").

appartiene(a,"C").

appartiene(b,"D").

questa volta, per ottenere la perfetta sovrapponibilità bisognerebbe legare a a c, ma a non è una variabile , è già legata a un preciso valore, vale appunto a, quindi questo tentativo fallisce. Si passa alla successiva clausola, quella indicata in rosso:

punto("A").

punto("B").

punto("C").

[1] Il termine *clausola* è usato, in generale, come sinonimo di *fatto* o di *regola*

punto("D").

punto("E").

retta(a).

retta(b).

retta(c).

retta(d).

appartiene(a,"B").

appartiene(c,"B").

→appartiene(a,"E").

appartiene(d,"D").

appartiene(b,"E").

appartiene(a,"C").

appartiene(b,"D").

Questa clausola soddisfa il goal legando X a "E", viene data notizia di questo e si prosegue nella ricerca di nuove soluzioni, ne viene ancora trovata una terza dove X viene legata a "C". Quando il Prolog ha percorso tutta la teoria il processo ha termine.

Vediamo che cosa succede quando il goal richiede l'intervento di clausole più complesse.Poniamo il seguente goal: incidenti(a,d).

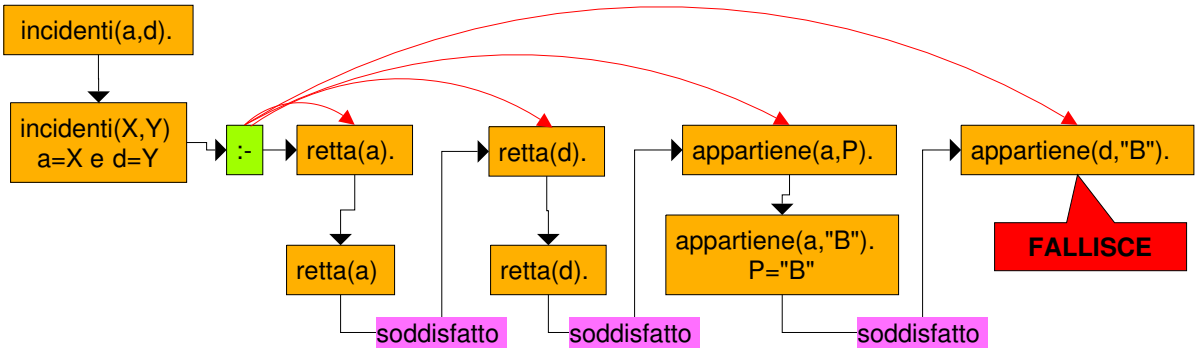

FIGURA 2.1.2. Diagramma

Il Prolog comincia a cercare nella base delle conoscenze un clausola che cominci con il predicato *incidenti* e trova la seguente clausola:

```
incidenti(X,Y) :-
  retta(X), retta(Y),
  appartiene(X,P),
  appartiene(Y,P).
```

Come riportato nella figura 2.1.2 nella pagina precedente, le variabili X e Y vengono legate rispettivamente alle costanti *a* e *d* , po, procedendo verso destra viene assunto il sottogoal *retta(a)* che è immediatamente soddisfatta dalla clausola omonima della teoria, successivamente, il Prolog, assume come sottogoal *retta(d).*, anch'esso immediatamente soddisfatto. Si passa poi al sottogoal *appartiene(a,P).*, scorrendo la teoria verso il basso si incontra la clausola *appartiene(a,"B").* che si sovrappone esattamente al nostro attuale sottogoal a patto di legare la variabile *P* con la costante *"B"*, da ultimo viene assunto come sottogoal appartiene(d,"B")., il Prolog, sempre partendo dall'alto della teoria verso il basso cerca un clausola da sovrapporre al sottogoal, ma non la trova, questo sottogaol fallisce. A questo punto il Prolog utilizza una tecnica chiamata *backtracking*[**1**, a pagina 86], ripercorre a ritroso il cammino compiuto fino ad incontrare l'ultima assegnazione di valore fatta a una variabile, cioè fino a raggiungere il punto in cui l'ultima variabile è stata legata, come è indicato nella figura 2.1.3

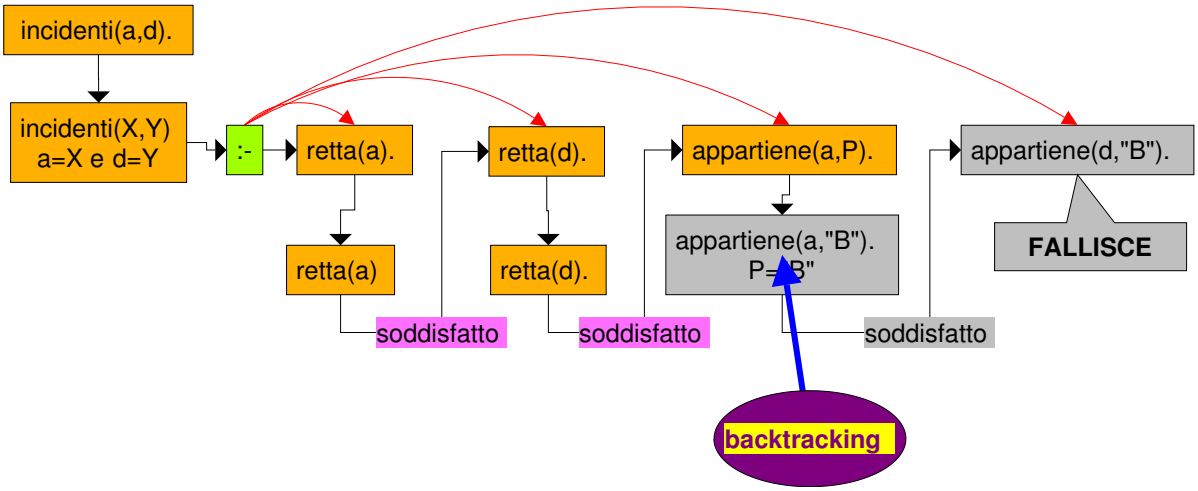

FIGURA 2.1.3. Diagramma 2

A questo punto si tenta di soddisfare il sottogoal *appartiene(a,P).* non partendo dall'inizio delle teoria, ma subito dopo alla clausola già utilizzata *appartiene(a,"B").* procedendo come è indicato nella figura: 2.1.4 nella pagina successiva

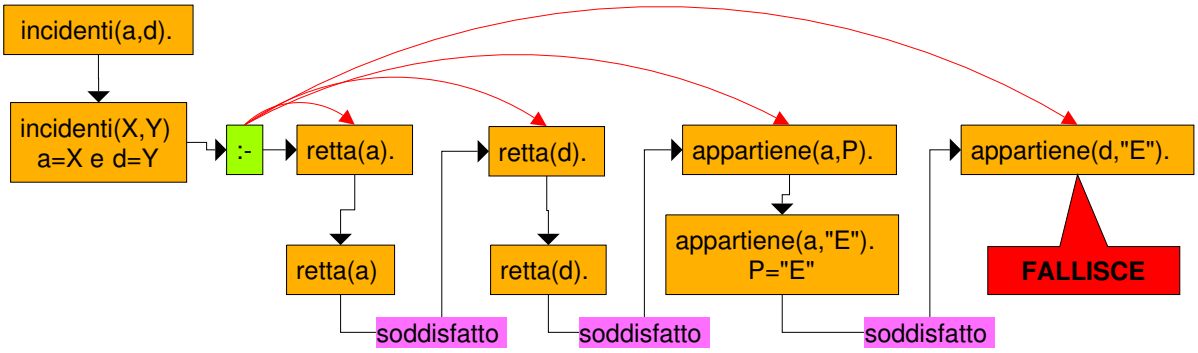

FIGURA 2.1.4. Diagramma 3

La variabile P viene legata alla costante "E", ma il tentativo di unificare *appartiene(d,"E").* a qualche clausola della teoria fallisce. Vi è quindi un secondo backtracking come il precedente ed un successivo terzo tentativo, durante il quale P viene legata a "C", ma l'unificazione di *appartiene(d,"C").* fallisce. A questo punto non è più possibile effettuare un backtracking come il precedente perché le clausole del tipo appartiene(a,....). sono finite, il backtracking deve risalire alla precedente clausola che ha legato le variabili X e Y, come è indicato in figura 2.1.5

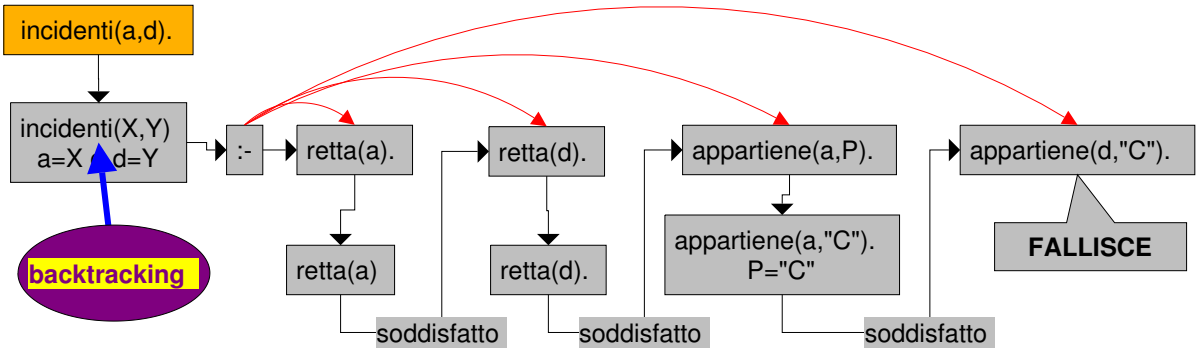

FIGURA 2.1.5. Diagramma 4

Le variabili X e Y vengono slegate ed il Prolog cerca una clausola avente il predicato incidenti(.....). successiva a quella già utilizzata, ma non la trova, di conseguenza il goal da noi proposto fallisce, il Prolog ne dà notizia come indicati in figura.

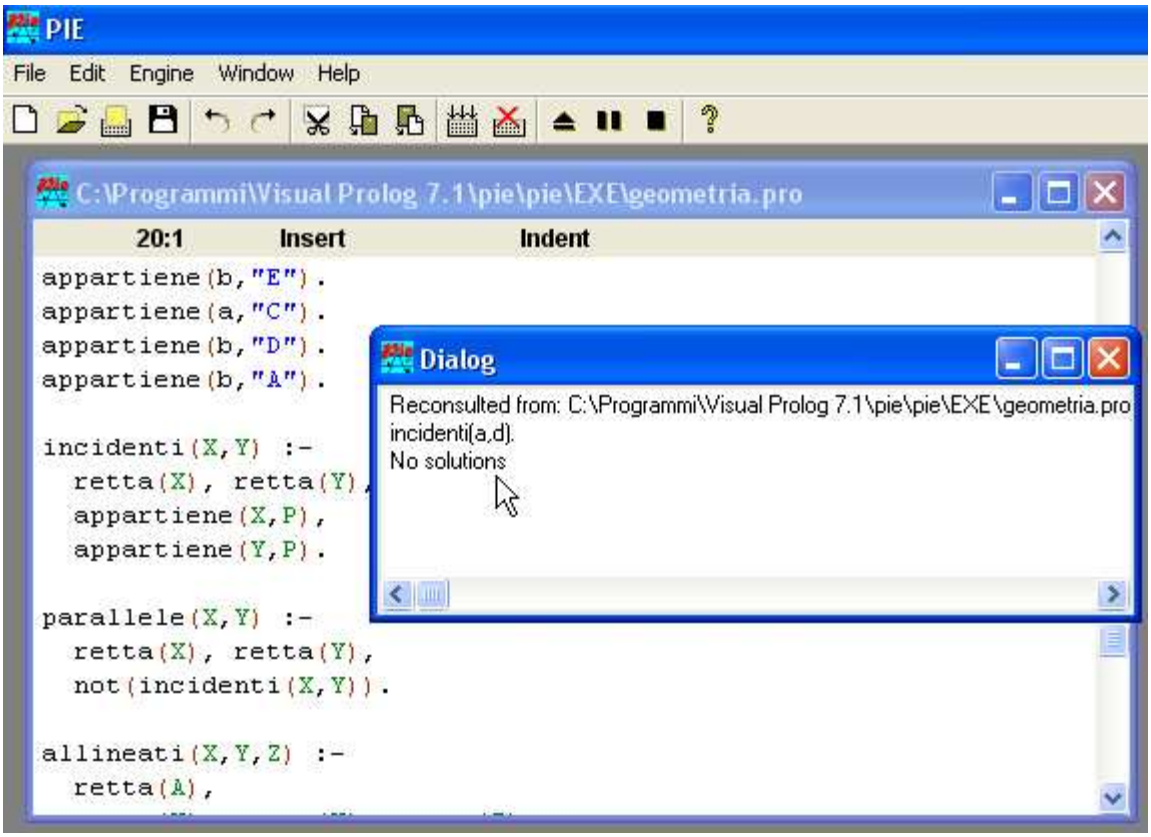

FIGURA 2.1.6.

2.2. Errori logici

Il programma da noi scritto vorrebbe simulare i tipici oggetti della geometria euclidea, quindi ci aspettiamo che si comportino come effettivamente si comportano gli oggetti di detta teoria. Ad esempio ci aspettiamo che alla domanda *incidenti(a,a).* il Prolog risponda *False*, invece guardate ciò che succede, figura

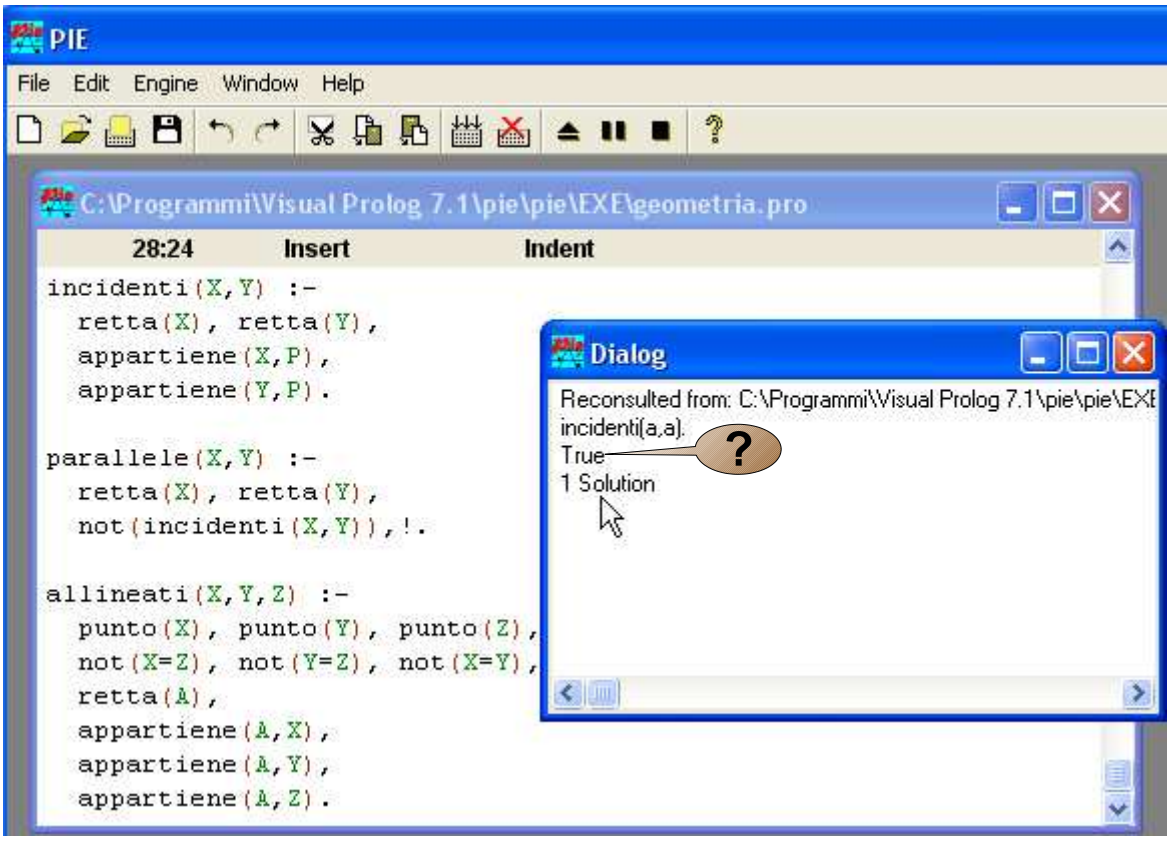

FIGURA 2.2.1.

Rileggendo la regola :

```
incidenti(X,Y):-
  retta(X), retta(Y),
  appartiene(X,P),
  appartiene(Y,P).
```

coi accorgiamo che, per il programma, due rette sono incidenti se hanno un punto in comune, e non come vorrebbe la definizione classica "due rette sono incidenti se hanno un punto e un solo punto in comune", ma anche se migliorassimo il programma togliendo questa incongruenza le cose non andrebbero bene. Infatti nella geometria euclidea ci sono infiniti punti e infinite rette, nel nostro modello possiamo dichiarare solo un mero finito di rette e di punti, potrebbe quindi succedere che una retta una sola volta come argomento di una clausola di tipo appartiene(.....), e quindi risulterebbe incidente con se stessa.

Tutto questo per segnalare che il Prolog non può garantire la coerenza logica dei modelli che vogliamo rappresentare nei nostri programmi, questa responsabilità è lasciata tutta al programmatore che scrive il programma.

2.3. Il CUT

Osserviamo il comportamento del seguente goal, figura 2.3.1

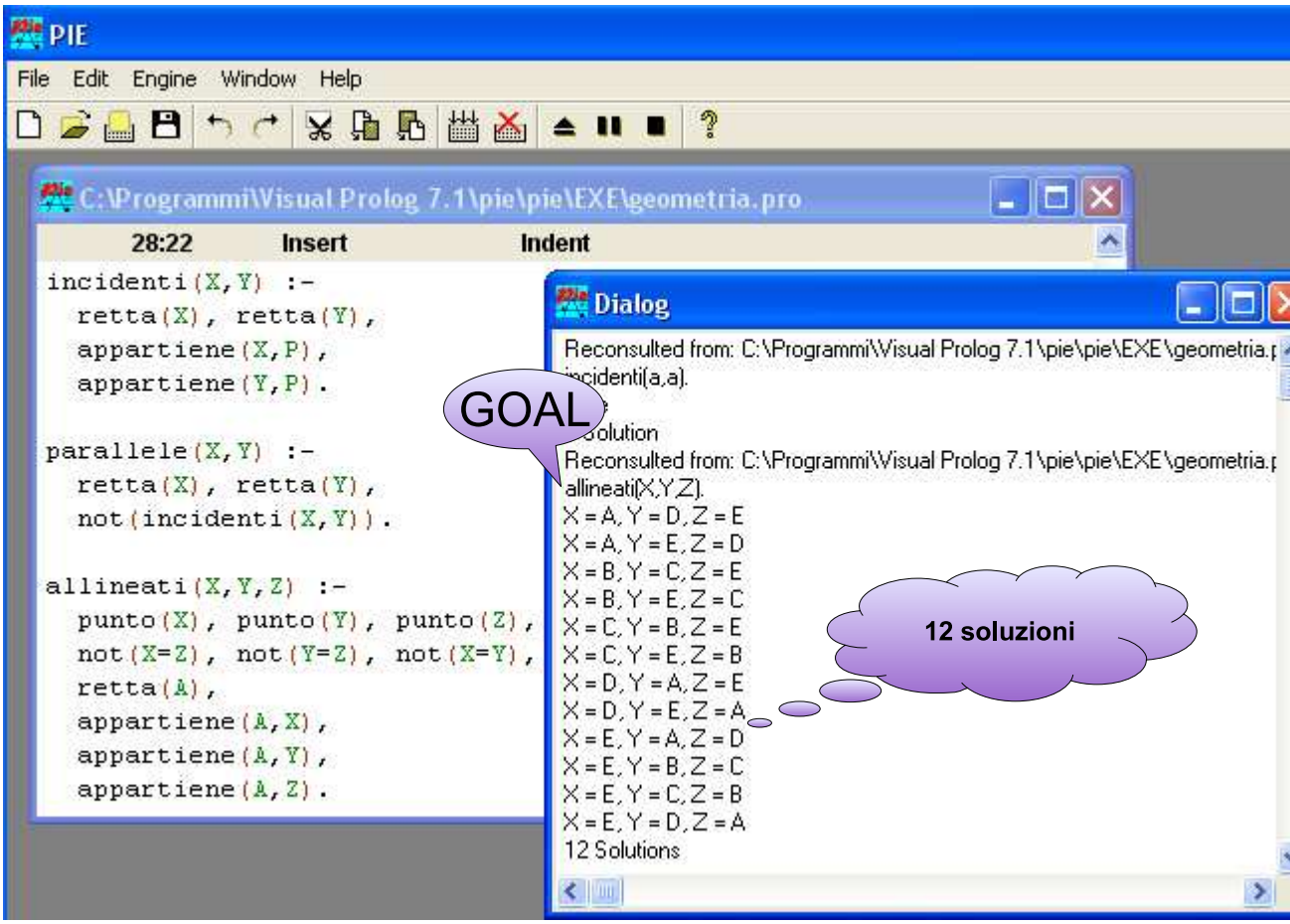

FIGURA 2.3.1. Il CUT

Il Prolog elenca tutte le possibili terne di punti allineati. In realtà la stesa terna viene ripetuta più volte, della stessa terna vengono date tutte le possibili permutazioni. In effetti, per come avviene il backtracking la terna ADE è diversa dalla terna DEA.

Proviamo ad aggiungere al programma il predicato predefinito `cut`, rappresentato dal punto esclamativo `!` nella posizione indicata nella figura 2.3.2 nella pagina seguente

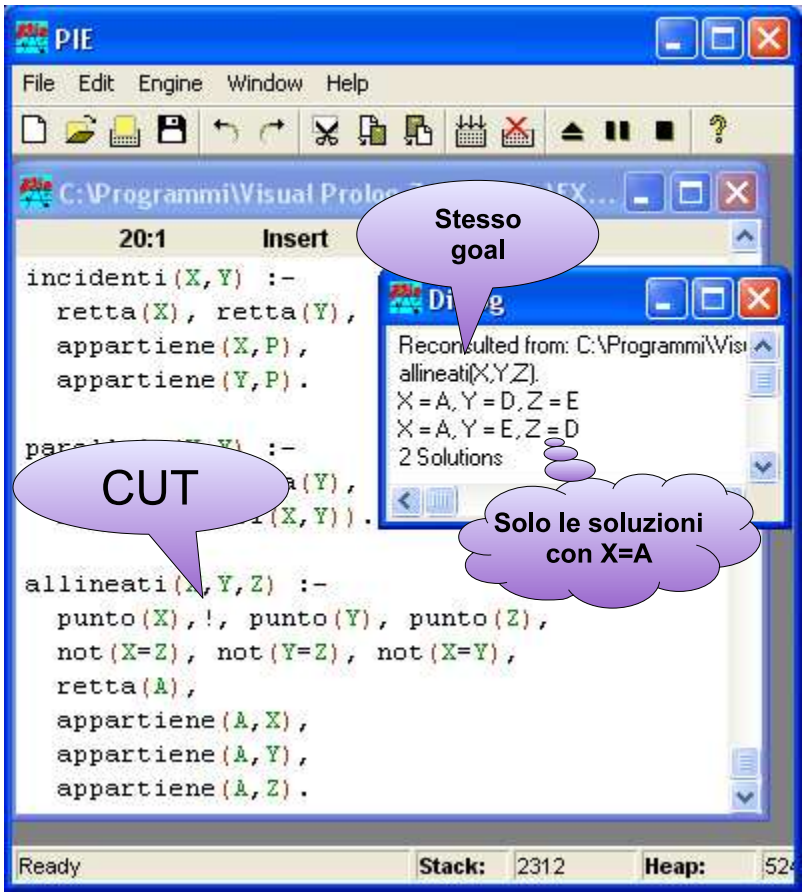

FIGURA 2.3.2. Il CUT

Immaginiamo che il Prolog realizzi il suo funzionamento tramite un folletto il quale svolazza sulle clausole del programma, ma seguendo sempre la stessa logica: parte sempre dall'inizio del programma e si muove verso destra e poi verso il basso. Il Prolog consegna al folletto il goal, questi, partendo dell'alto cerca una clausola la cui testa si possa unificare con la testa del goal. Trovata la clausola che va bene si muove verso destra ed unifica eventualmente delle variabili e magari incontra un sottogoal. Si segna la posizione del sottogoal, abbandono momentaneamente il goal principale ed assume come nuovo goal il sottogoal incontrato. Ripartendo dall'inizio cerca di soddisfare, con la stessa logica, questo nuovo goal, se non ci riesce tutto fallisce, se ci riesce riprende il vecchio goal e precede prima a destra fino a quando non ha soddisfatto tutti i sottogoal incontrati. Se non riesce a soddisfare tutti i sottogoal allora avviene il backtracking, cioè il folletto torna indietro, muovendosi verso sinistra, fino al punto in cui ha legato l'ultima variabile incontrata, la slega e riprende il cammino cercando di soddisfare il sottogaol che aveva legato la variabile come indicato nella sezione 2.1 nella pagina 21 . Il CUT è una clausola strana, ha sempre successo, per il folletto è come una valvola che lo lascia passare solo in una direzione, da sinistra verso destra, non può più riattraversarlo per fare il backtracking, inoltre tutte le variabili legate tra la testa della clausola e al

momento dell'attraversamento vengono congelate e non possono più essere slegate, quindi se per per il successo del goal attuale fosse necessario un backtracking su queste variabili il goal attuale fallirebbe.

Il CUT ha un comportamento che dipende dall'ordine con cui sono state scritte le clausole. Proviamo a cambiare il loro ordine e riproporre lo stesso goal della figura 2.3.2 a fronte, ecco che cosa succede (vedi figura 2.3.3)

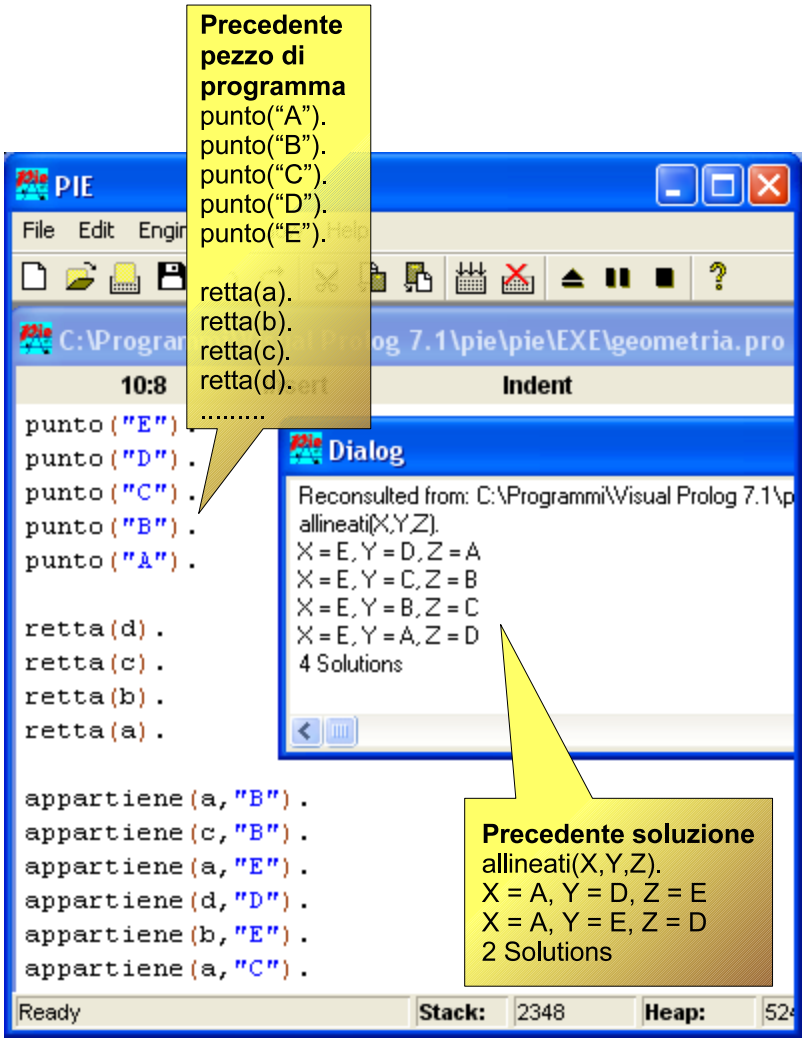

FIGURA 2.3.3.

Le soluzioni sono completamente diverse da quelle riportate nella figura 2.3.2 a fronte. La presenza del CUT fa in modo che le soluzioni dipendano dall'ordine con cui sono scritte le clausole, mentre un programma senza CUT produce soluzioni che sono indipendenti, a parte l'ordine, dall'ordine delle clausole nel programma come si può vedere nella figura 2.3.4 nella pagina successiva.

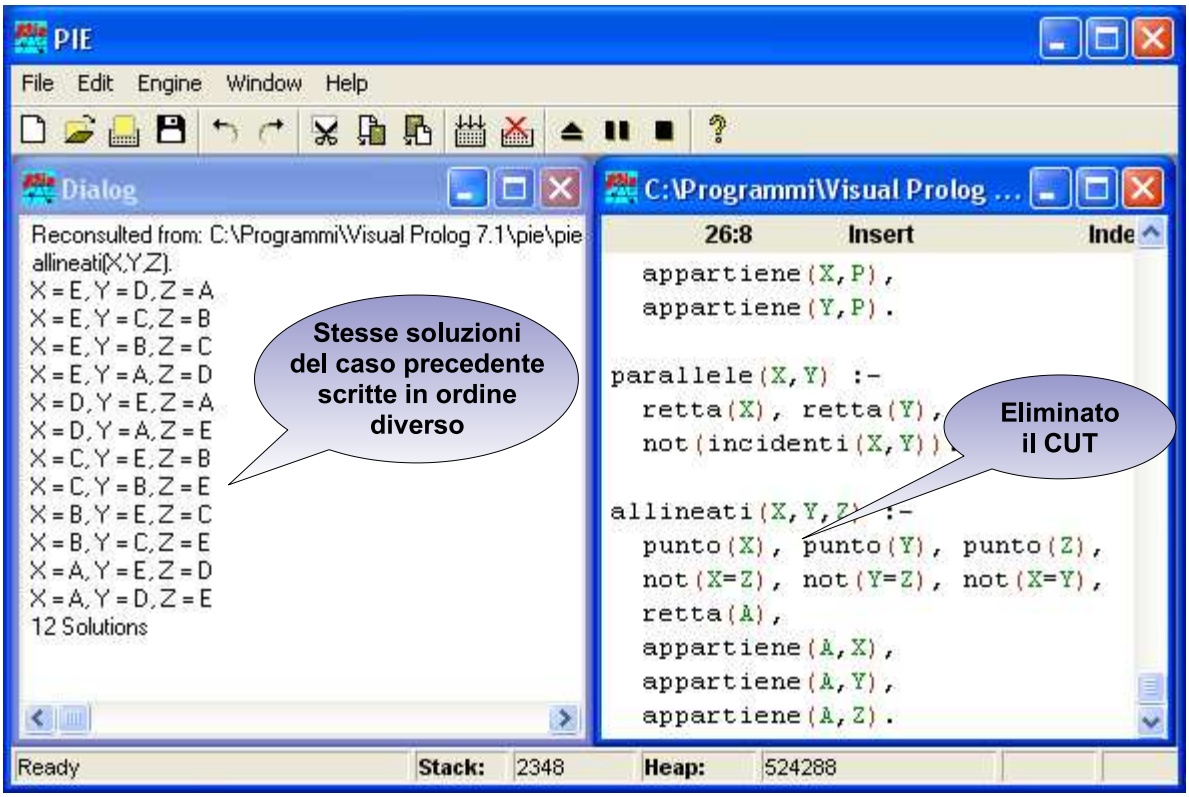
PIE
File Edit Engine Window Help
Dialog
Reconsulted from: C:\Programmi\Visual Prolog 7.1\pie\pie
allineati(X,Y,Z).
X = E, Y = D, Z = A
X = E, Y = C, Z = B
X = E, Y = B, Z = C
X = E, Y = A, Z = D
X = D, Y = E, Z = A
X = D, Y = A, Z = E
X = C, Y = E, Z = B
X = C, Y = B, Z = E
X = B, Y = E, Z = C
X = B, Y = C, Z = E
X = A, Y = E, Z = D
X = A, Y = D, Z = E
12 Solutions
Stesse soluzioni del caso precedente scritte in ordine diverso
C:\Programmi\Visual Prolog ...
26:8 Insert Inde
appartiene(X,P),
appartiene(Y,P).
parallele(X,Y) :-
retta(X), retta(Y),
not(incidenti(X,Y)).
allineati(X,Y,Z) :-
punto(X), punto(Y), punto(Z),
not(X=Z), not(Y=Z), not(X=Y),
retta(A),
appartiene(A,X),
appartiene(A,Y),
appartiene(A,Z).
Eliminato il CUT
Ready Stack: 2348 Heap: 524288

FIGURA 2.3.4.

2.4. Le Liste

La lista è una sequenza di oggetti racchiusi tra parentesi quadre. Ecco alcuni esempi di liste:

Lista_1: [bianco,rosso,azzurro,celeste]
Lista_2: [1,112,2E32,3.5E-4]
Lista_3: ["Giorgio","Renata","Lucio","Ester"]

Esiste anche la lista vuota, quella che non contiene elementi, la si indica in questo modo []. Le liste hanno una loro struttura interna.

$$\left[\underbrace{bianco}_{TESTA}, \underbrace{rosso, azurro, celeste}_{CODA}\right] \tag{2.4.1}$$

le liste sono formate da una testa, costituita dal primo elemento della lista e da una coda formata dalla lista costituita dai rimante elementi della lista di partenza. La suddivisione di una lista in testa e coda è esattamente il modo di operare del Prolog quando incontra una lista, la coda di una lista è sempre considerata a sua volta una lista che può essere eventualmente vuota. Quando usiamo delle variabili all'interno delle liste dobbiamo mettere in evidenza la struttura interna della lista in questo modo

$$[Testa|Coda]$$

Per illustrare meglio questo concetto ricorriamo al seguente esempio (vedi figura 2.4.1)

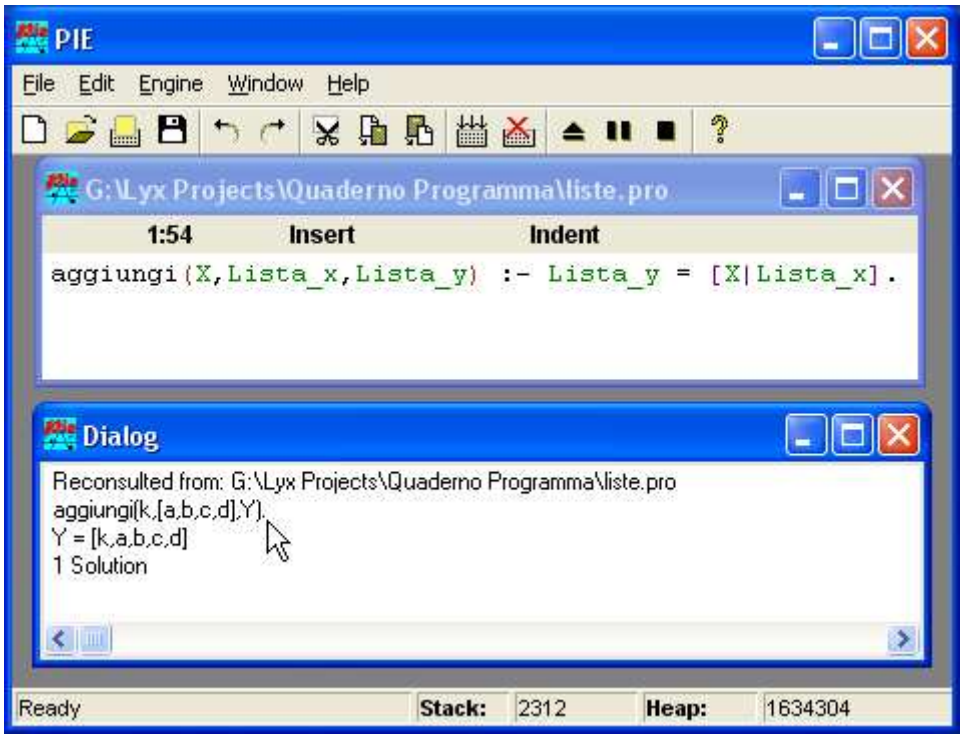

FIGURA 2.4.1.

Ponendo il goal *aggiungi(k,[a,b,c,d],Y).*, nell'unica clausola del programma avviene che:

- la variabile X viene legata a k
- la lista Lista_x viene legata alla lista [a,b,c,d]
- si assume come nuovo goal Y=[k|Lista_x]. Lista_x è già stata legata a [a,b,c,d], quindi viene formata la lista che ha come testa k e come coda Lista_x, successivamente l'operatore = cerca di unificare le variabili ancora rimaste libere in modo che il primo membro sia identico al secondo membro, di conseguenza Y viene legata alla lista [k,a,b,c,d] e il goal ha successo.

2.5. La Ricorsione

Una clausola si dice ricorsiva quando nel corpo della clausola vi è un predicato che coincide con quello della testa, cioè è una clausola che richiama se stessa. Per illustrare questo concetto costruiamo un programma che conta gli elementi di una lista (vedi figura 2.5.1)

```
conta(0,[]).
conta(X,[Testa|Coda]) :- conta(Y,Coda), X is Y+1.
```

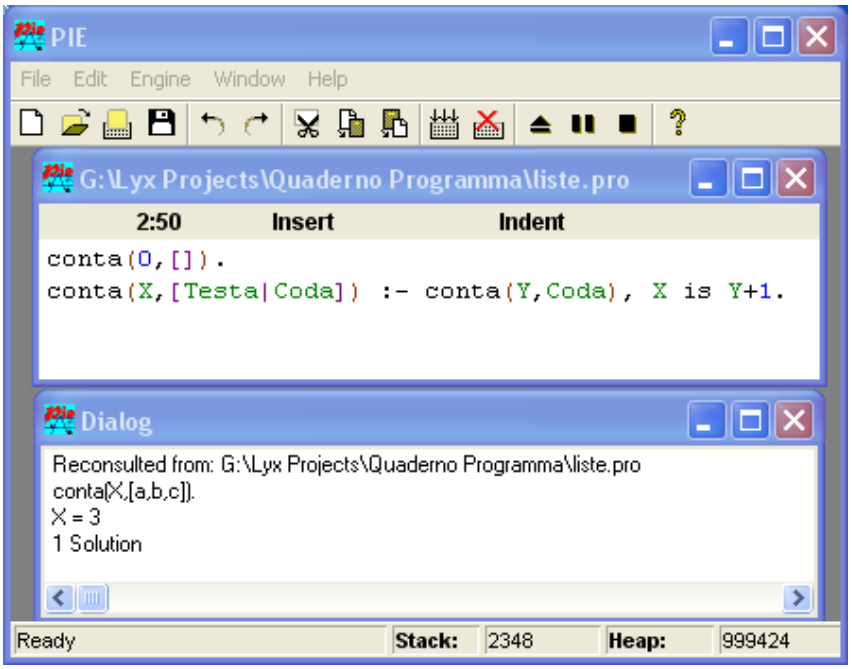

FIGURA 2.5.1. Clausole ricorsive

Nella figura 2.5.2 a fronte è indicato il funzionamento del programma

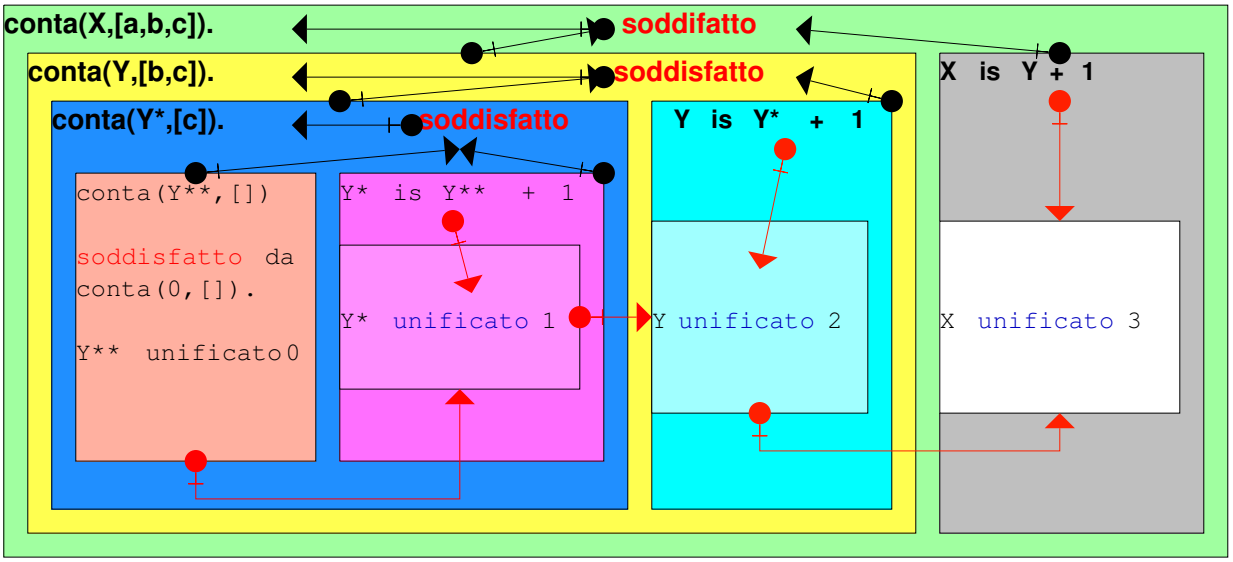

FIGURA 2.5.2. Diagramma di una clausola ricorsiva

Partendo dal goal *conta(X,[a,b,c]).* viene richiamata la seconda clausola del programma con la variabile *X* libera, *Testa* unificata ad *'a'* e *Coda* unificata alla lista *[b,c]*. Il goal attuale da soddisfare è dunque quello racchiuso nel rettangolo verde .Per soddisfare il goal verde bisogna soddisfare prima *conta(Y,[b,c]).* (goal giallo), che diventa il goal attuale) e poi *X is Y+1* (goal grigio). Per soddisfare il goal giallo bisogna soddisfare *conta(Y*,[c]).* (goal blu)[2] e il goal *Y is Y*+1* (goal celeste). Il goal attuale diventa quindi il goal blu per soddisfare il quale bisogna soddisfare *conta(Y**,[]).* (goal arancio), questo goal diventa goal attuale e viene soddisfatto dalla prima clausola del programma la quale unifica Y** a 0. Si passa quindi a soddisfare il goal *Y* is Y**+1* (goal fucsia) che diventa attuale, per soddisfarlo *Y** viene unificata a *Y**+1*, cioè a *0+1*, quindi a 1. Il goal blu a, a questo punto è soddisfatto con $Y* \rightarrow 1$. Successivamente diventa attuale il goal celeste, per soddisfarlo *Y* viene unificata a Y*+1 cioè si ha $Y \rightarrow 2$. Il goal giallo è così soddisfatto, si passa quindi al goal grigio, che diventa attuale, per soddisfarlo *X* viene unificata a *Y+1*, quindi $X \rightarrow 3$. A questo punto anche il goal verde è soddisfatto quindi viene stampato *X=3*.

Da osservare l'uso della clausola *is*, si tratta di un operatore infisso che funziona nel seguente modo:

$$X \quad is \quad Espressione(Y, Z, W). \tag{2.5.1}$$

viene valutata matematicamente $Espressione(Y, Z, W)$, successivamente la variabile *X* viene unificata al risultato di detto calcolo, naturalmente, al momento della valutazione dell'espressione le variabili *Y* , *Z* , *W* devono essere legate, mentre X deve essere libera.

[2]Siccome viene richiamata sempre conta(X,[Testa|Coda])].....dovremmo sempre usare X e Y, ma esse hanno un ruolo diverso che dipende dal goal attuale, abbiamo quindi usato gli asterischi per sottolineare questa diversità.

CAPITOLO 3

Le classi

3.1. Il Programma calcolo_fattoriale

Abbandoniamo P.I.E. e ritorniamo ad usare Visual Prolog. Visual Prolog è un linguaggio orientato agli oggetti per costruire i quali fa uso delle classi. Il concetto di classe è comune a tutti i linguaggi di questo tipo, ma ha implementazioni diverse nei vari linguaggi. Per illustrare che cosa succede nel Visual Prolog cominciamo con alcuni esempi.

Utilizzando Project ⟶ New compare la solita form nella quale scegliete la cartella *General* e la riempite come sotto indicato

```
General
   Project Name: calcolo_fattoriale
   UI Strategy: Object-oriented GUI(pfc/gui)
   Target Type: Exe
   Base Directory: C:\.......(fate voi)
```

Compiliamo il tutto ed otteniamo un'applicazione vuota. Aggiungiamo al progetto un nuovo package che chiameremo *contenitore_form*. Per fare questo *scegliamo New come è indicato nella figura 3.1.1*

FIGURA 3.1.1. New package

comparirà una form da compilare come è indicato nella figura 3.1.2 nella pagina seguente

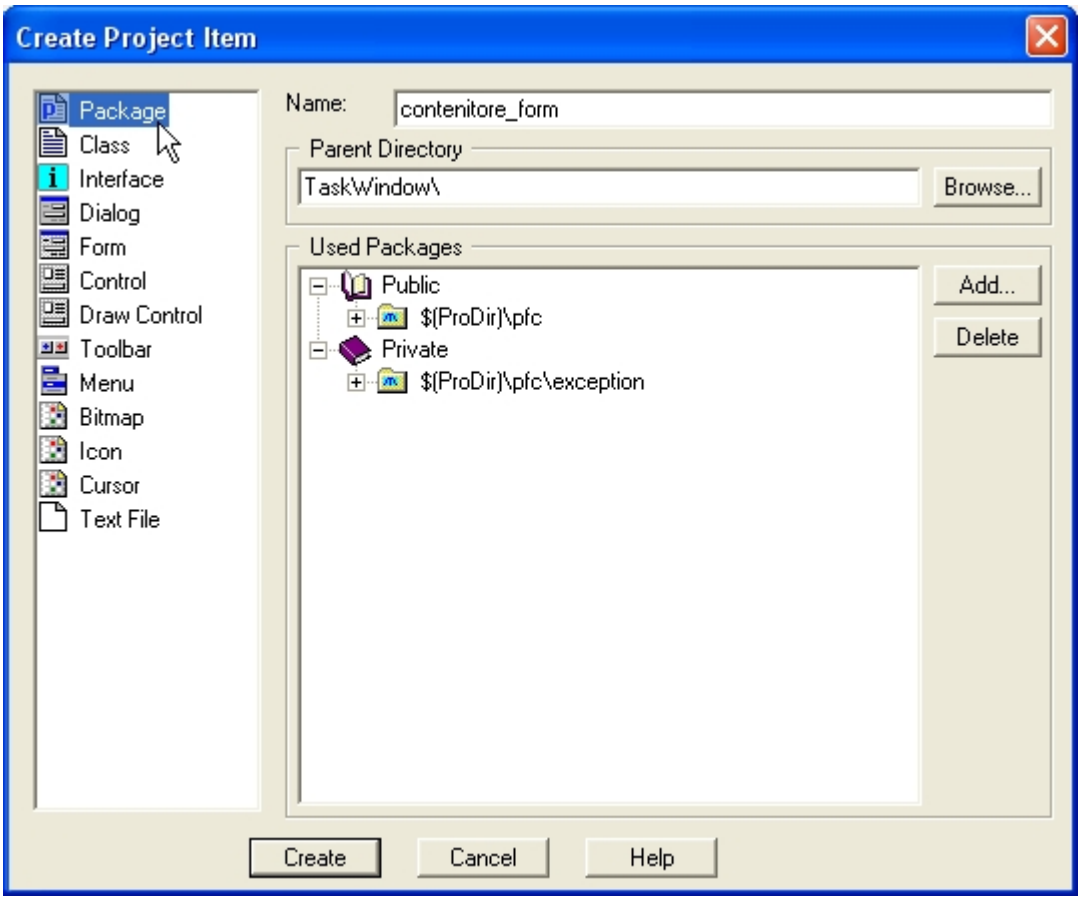

FIGURA 3.1.2. Nuovo package

scegliendo Crea otteniamo ,il nuovo package Contenitore_form .

Inseriamo nel nuovo package una form alla quale daremo il nome di *inserimento_dati* , innanzitutto bisogna scegliere, nell'albero del progetto, il package come indicato in figura 3.1.3

FIGURA 3.1.3. Scelta del package

successivamente con la combinazione New ⟶ form otteniamo la seguente finestra di dialogo da riempire come indicato nella figura 3.1.4

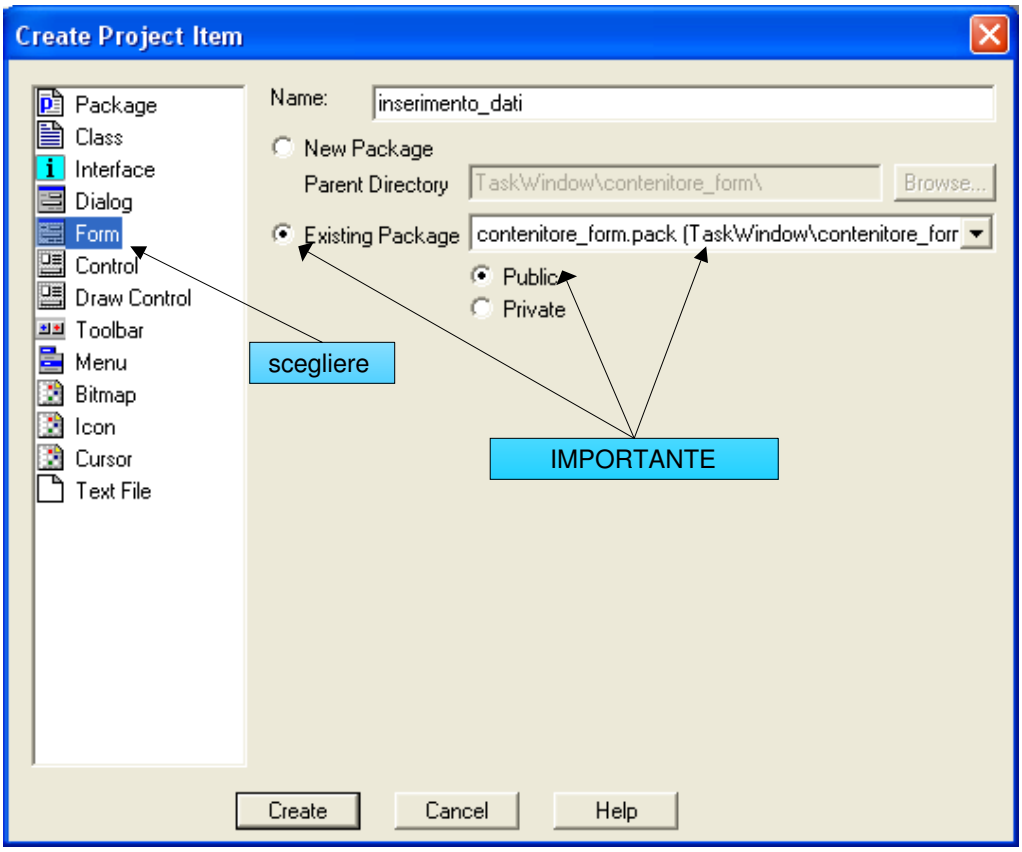

FIGURA 3.1.4. Creare una nuova form

scegliendo, infine, Create otteniamo la form desiderata. Aggiungiamo alla form un pulsante e un campo di testo come indicato nella figura 3.5.4 nella pagina 99

Adesso bisogna introdurre delle funzionalità che utilizzino i dati introdotti con la form inserimento_dati, per fare questo utilizzeremo un nuovo costrutto chiamato classe. Per introdurre una classe nel progetto si procede in modo simile a quanto fatto per creare la form inserimento_dati, scegliete View ⟶ Project Window, aprite TaskWindow e scegliete contenitore_form, come indicato nella figura 3.1.6 nella pagina 41 (1° click), successivamente andate sull'icona New (figura 3.1.6 nella pagina 41 2° click), si apre la form *Create Project Item*, scegliete Class (figura 3.1.6 nella pagina 41 3° click), riempite il campo Name come in figura e scegliete l'opzione Existing package (figura 3.1.6 nella pagina 41 4° click), il relativo campo è già compilato, infine scegliete Create ed avrete la vostra classe.

Se compilate il progetto otterrete una applicazione che non fa nulla di diverso rispetto a prima, cioè otterrete ancora un'applicazione vuota. Questo perché il programma è in grado di manipolare nuovi oggetti, ma questi non sono stati creati e quindi attualmente

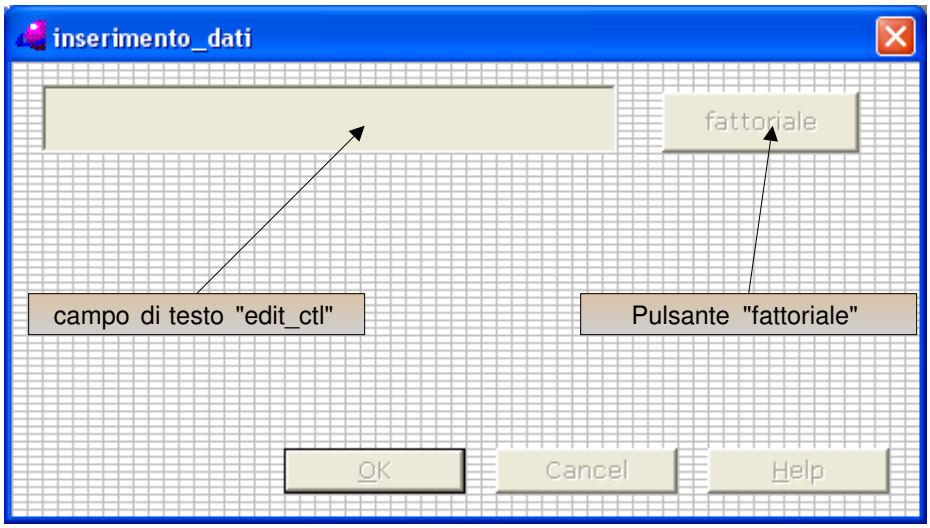

FIGURA 3.1.5. La form completa

non esistono. Le classi sono dei tipi di dati, un po' più complessi rispetto al tipo integer, real, symbol etc, mentre gli oggetti sono i dati veri e propri, potremo quindi creare oggetti di tipo inserimento_dati oppure di tipo fatt, bisogna però scrivere del codice affinché ciò accada. Osserviamo come nella finestra Project Window le cose siano cambiate ci sono dei collegamenti a nuovi file che prima non c'erano.

Cerchiamo di dare delle funzionalità al programma , innanzitutto vorremmo che *New* della barra delle icone e quello della barra dei menu producessero una form di tipo *inserimento_dati* e la visualizzassero. Per fare questo dal menu View scegliamo Project Window come indicato nella figura 3.1.7 nella pagina 42

Nella finestra Project Window scegliamo TaskMenu.mnu facendo click con il tasto destro compare un pop up nel quale scegliamo Edit , comparirà la finestra riportata in figura 3.1.8 nella pagina 43, apriamo File e scegliamo New , togliamo il segno di spunta alla casella Disabled

otteniamo così la situazione descritta nella figura 3.1.9 nella pagina 44. Salviamo il tutto e ricompiliamo, lanciando l'applicazione ci accorgeremo che nel menu *File* la voce *New* è ora abilitata.

Ritorniamo alla finestra Project Window nella quale sceglieremo TaskWindow aprendo il quale possiamo scegliere TaskWindow.win, facendo click con il tasto destro comparirà il menù indicato in figura 3.1.10 nella pagina 45 e a questo punto scegliamo Code Expert

La scelta di Code Expert visualizza la finestra Dialog and Window Expert nella quale noi apriremo Menu ⟶ TaskMenu ⟶ id_file ottenendo quanto riportata nella figura 3.1.11 nella pagina 46.

Scegliamo Add e otteniamo i cambiamenti riportati nella figura 3.1.12 nella pagina 47

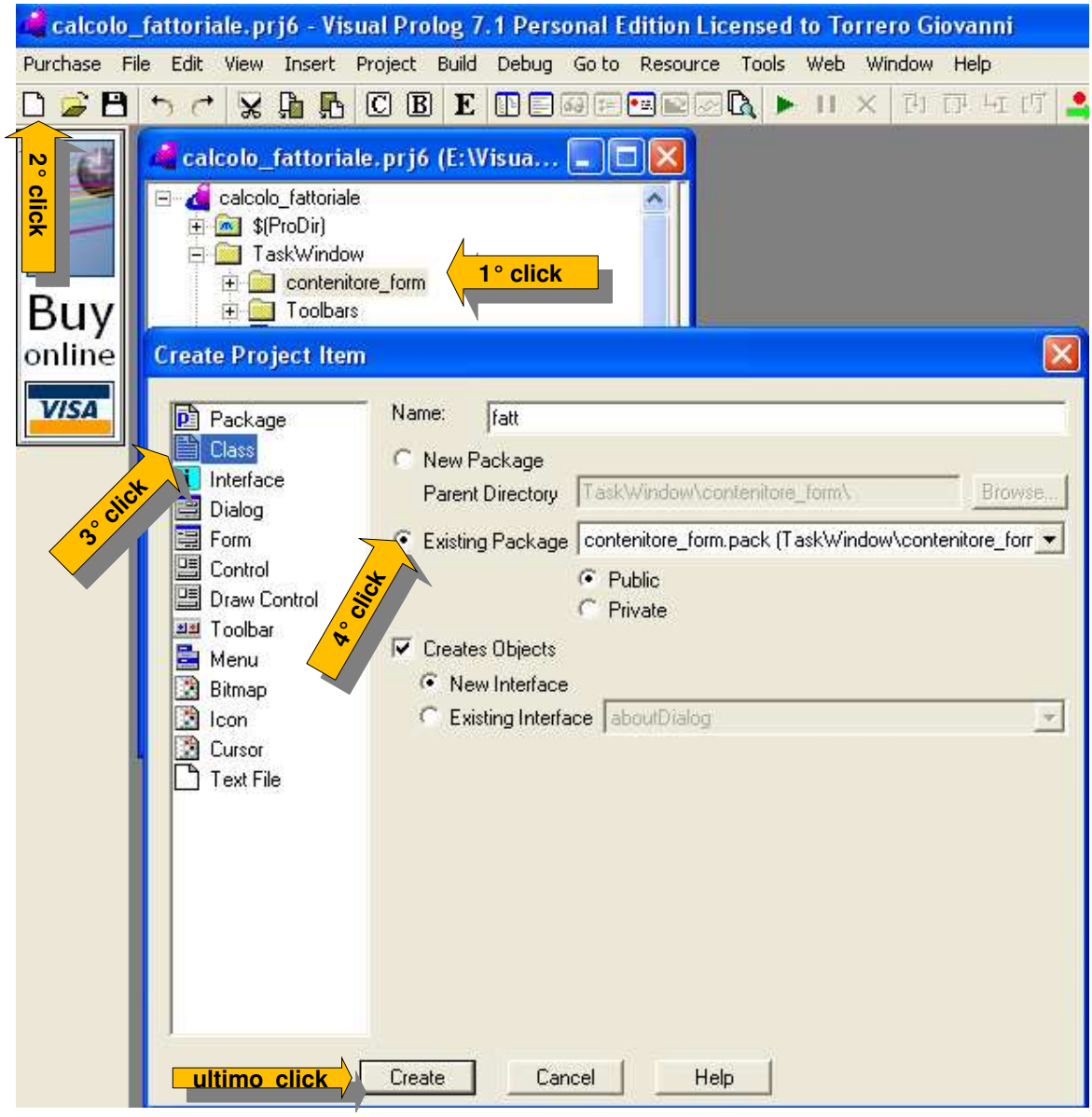

FIGURA 3.1.6. Costruire una classe

Se facciamo doppio click come indicato nella figura 3.1.12 nella pagina 47 veniamo catapultati nella finestra TaskWindow.pro. Le righe di codice che precedono la parte evidenziate sono state scritte da Code Expert.

Il frammento di codice evidenziato va modificato come è indicato in figura 3.1.14 nella pagina 49. Salvate il tutto e compilate. Eseguendo l'applicazione vi accorgerete che facendo click su New appare la form inserimento_dati il cui pulsante fattoriale è però privo di ogni funzionalità.

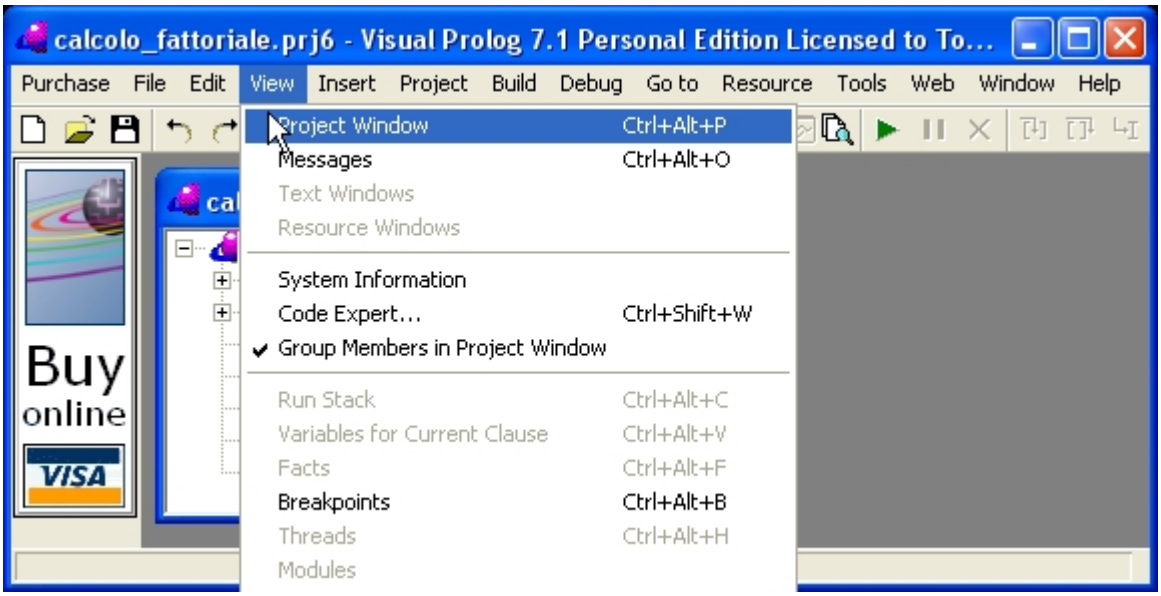

FIGURA 3.1.7. Primo passo

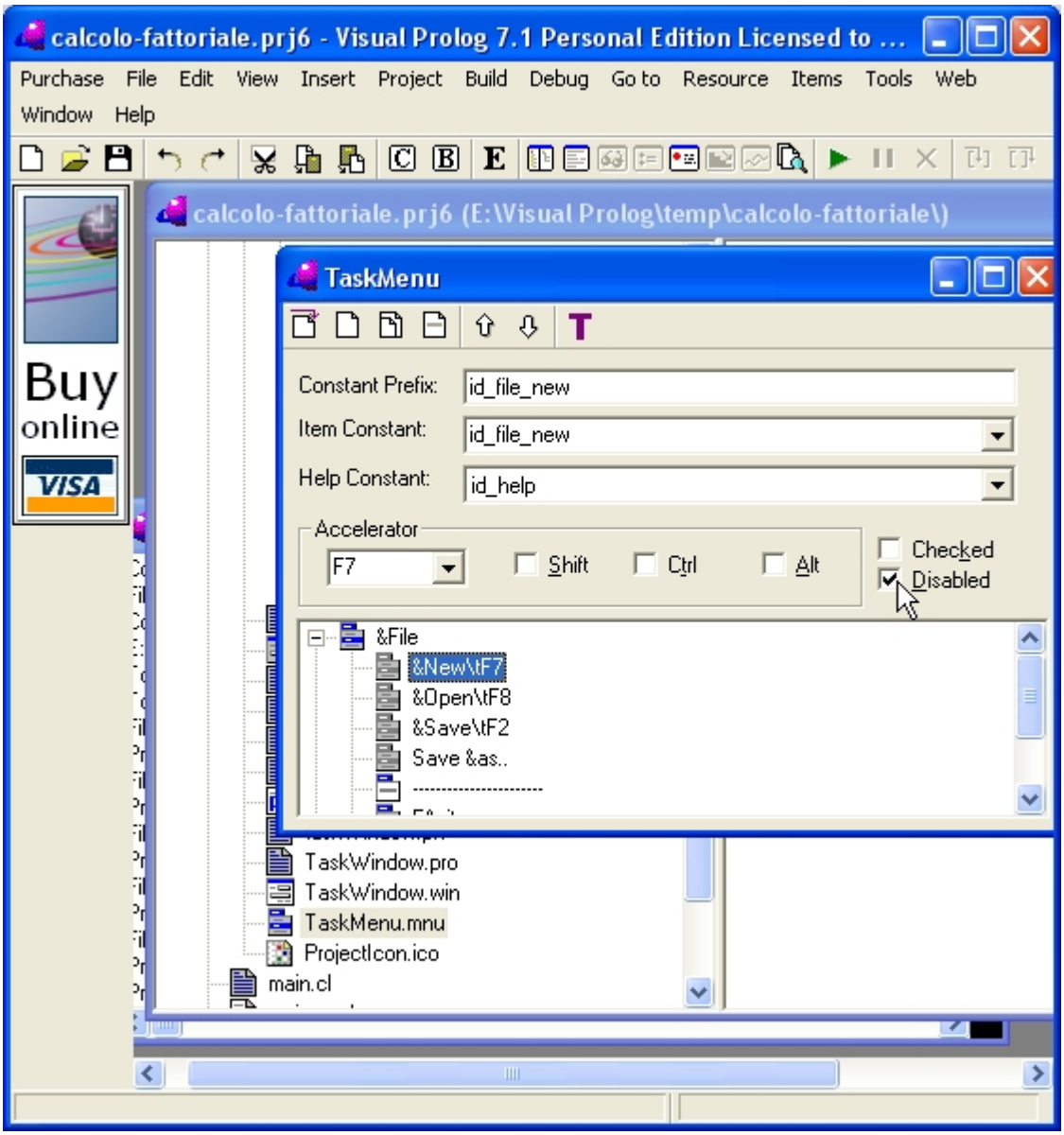

FIGURA 3.1.8. Abilitiamo New

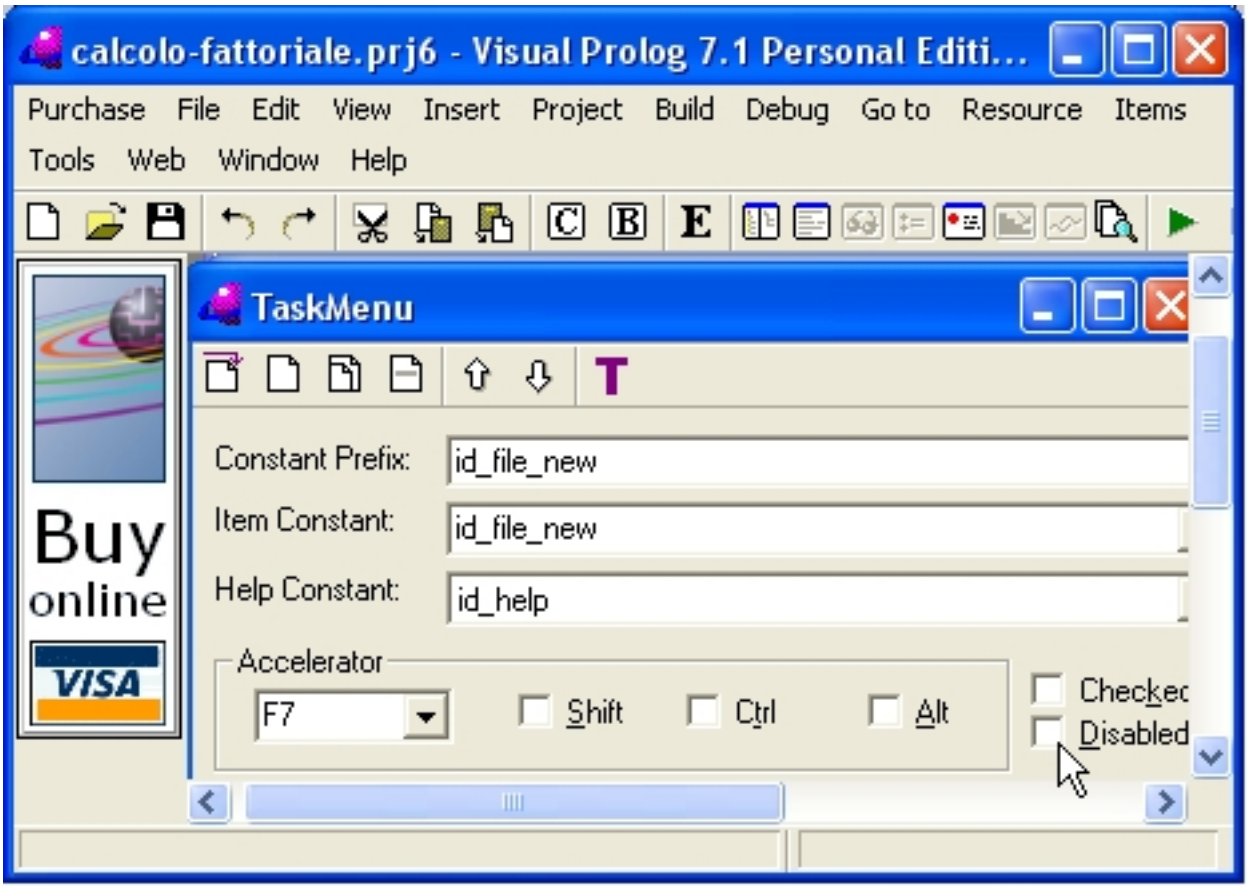

FIGURA 3.1.9. Abilitiamo New 2

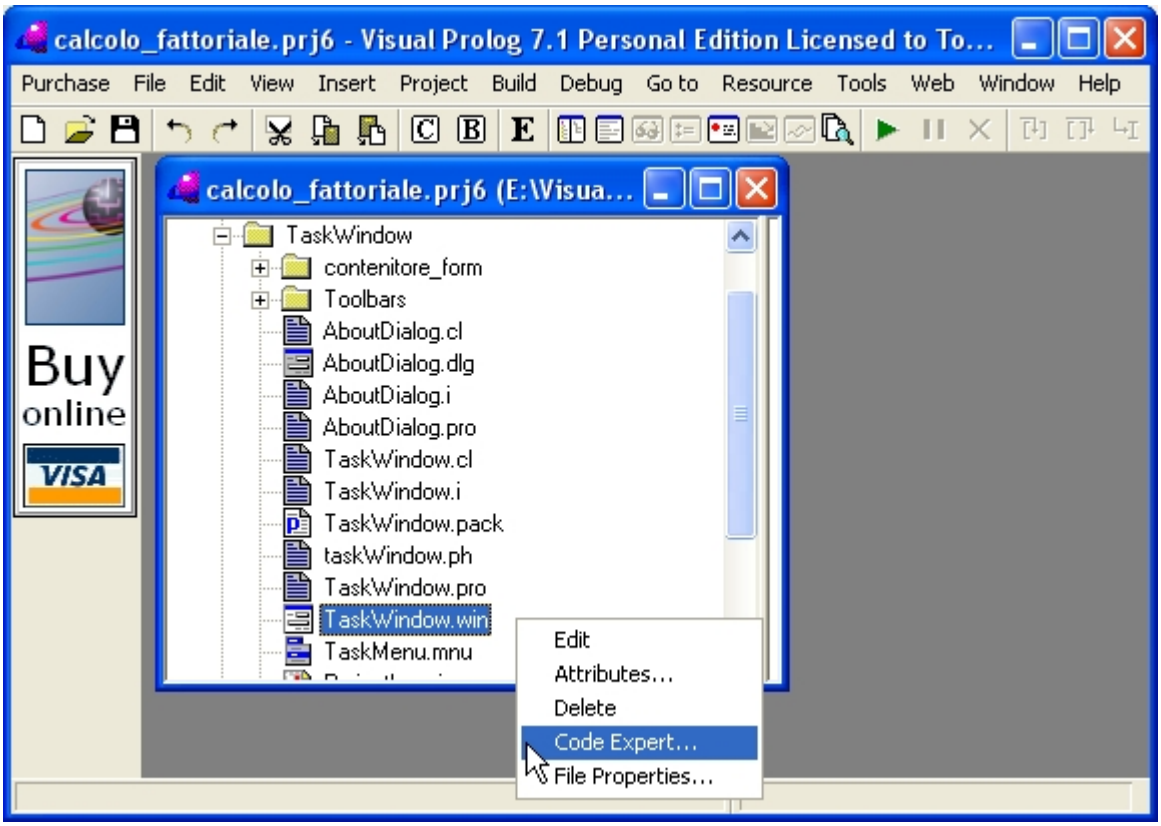

FIGURA 3.1.10. Secondo passo

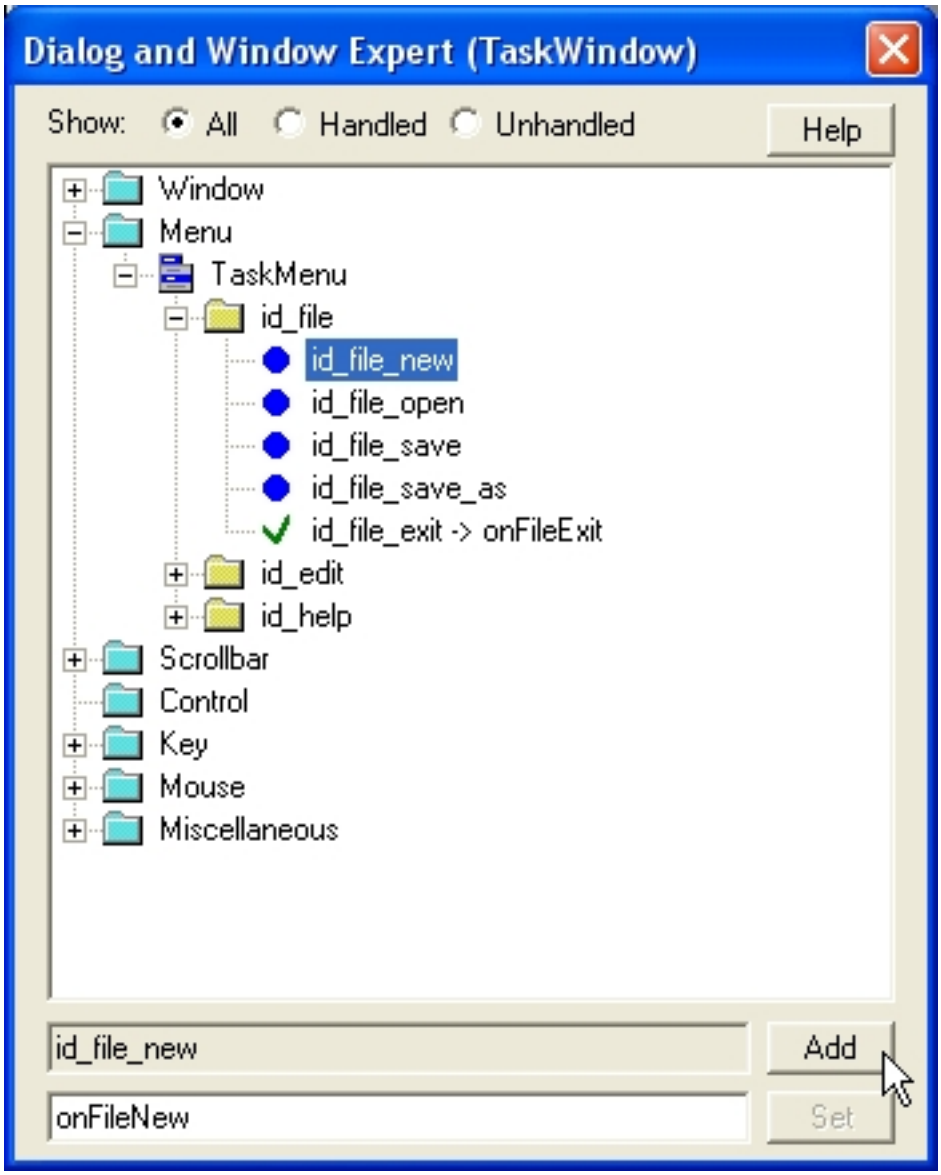

FIGURA 3.1.11. Window Expert

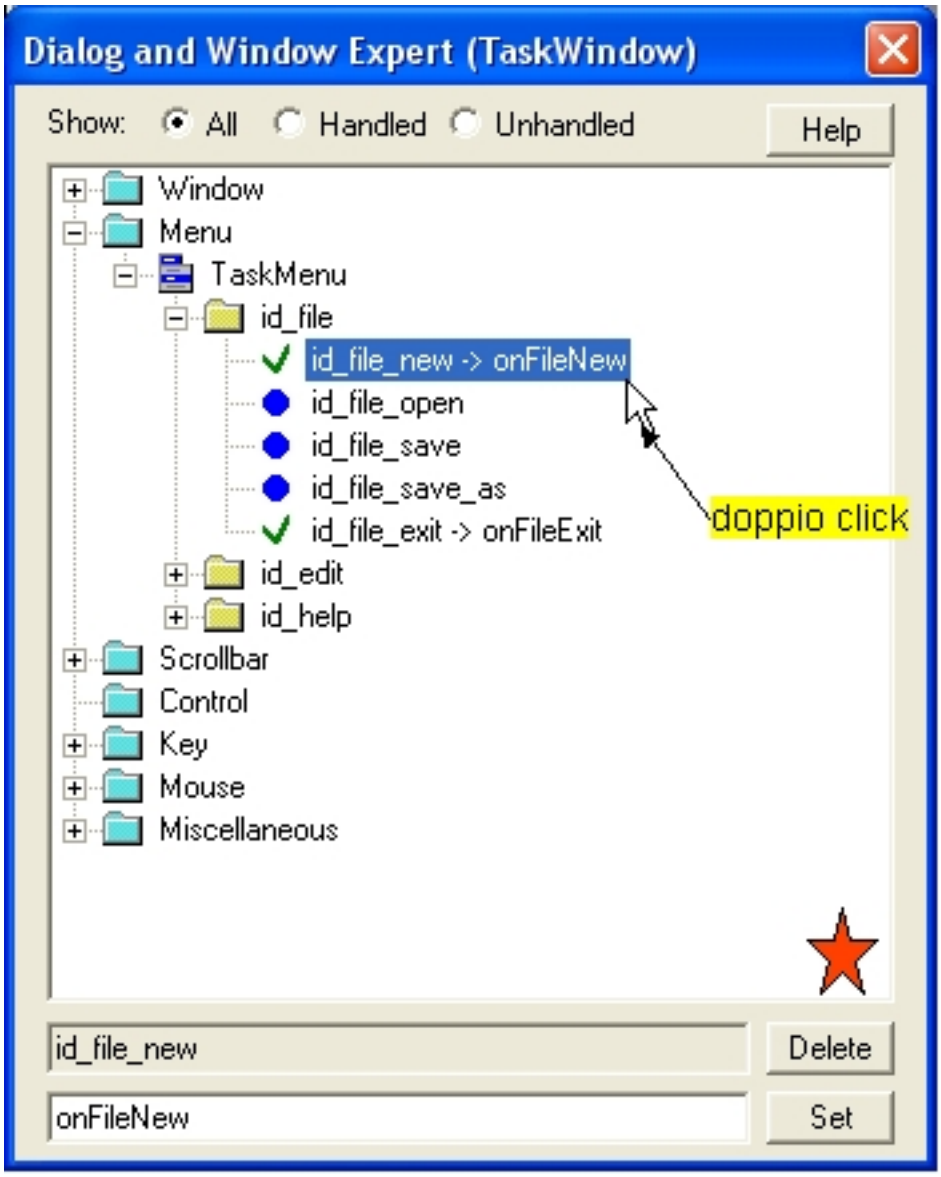

FIGURA 3.1.12. Window Expert 2

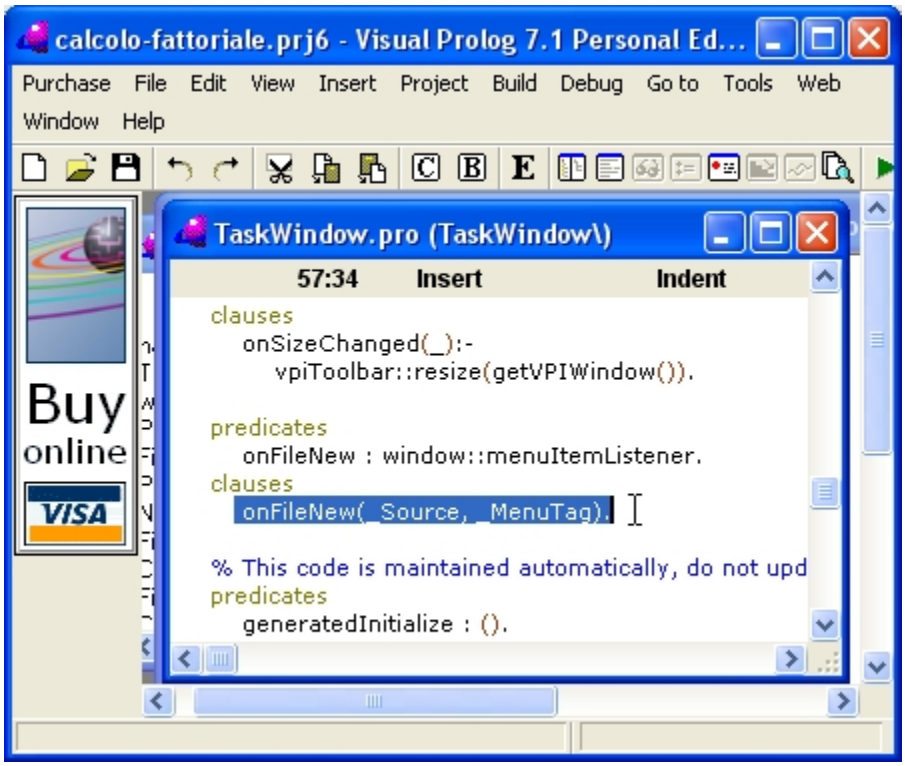

FIGURA 3.1.13. TaskWindow.pro-1

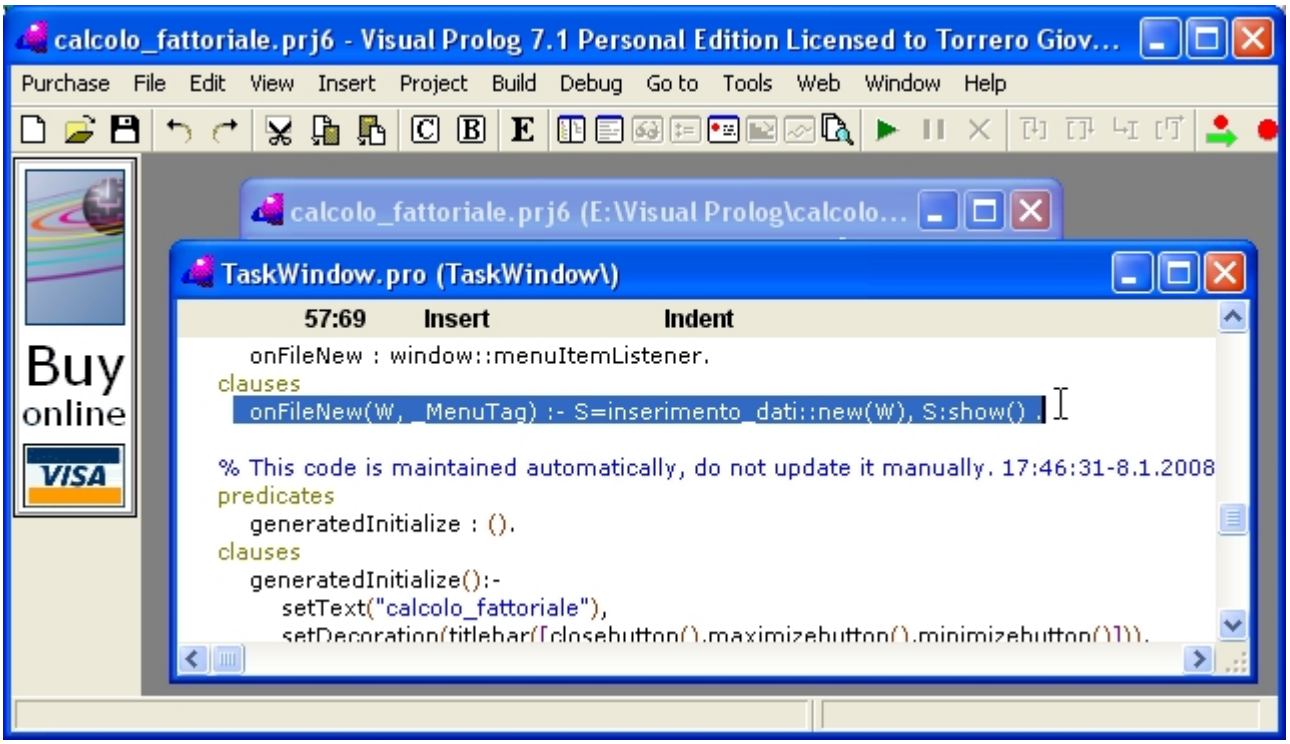

FIGURA 3.1.14. TaskWindow.pro-2

Per dare funzionalità al pulsante *fattoriale* utilizziamo la classe *fatt*. Il Visual Prolog ha una procedura automatica per creare le classi, questa procedura costruisce lo scheletro della classe, cominciamo ad analizzare questo scheletro.

Per creare una nuova classe si sceglie New dal menu File oppure direttamente dalla barra dei comandi si sceglie l'icona , compare la finestra di dialogo riportata nella figura 3.1.15 .

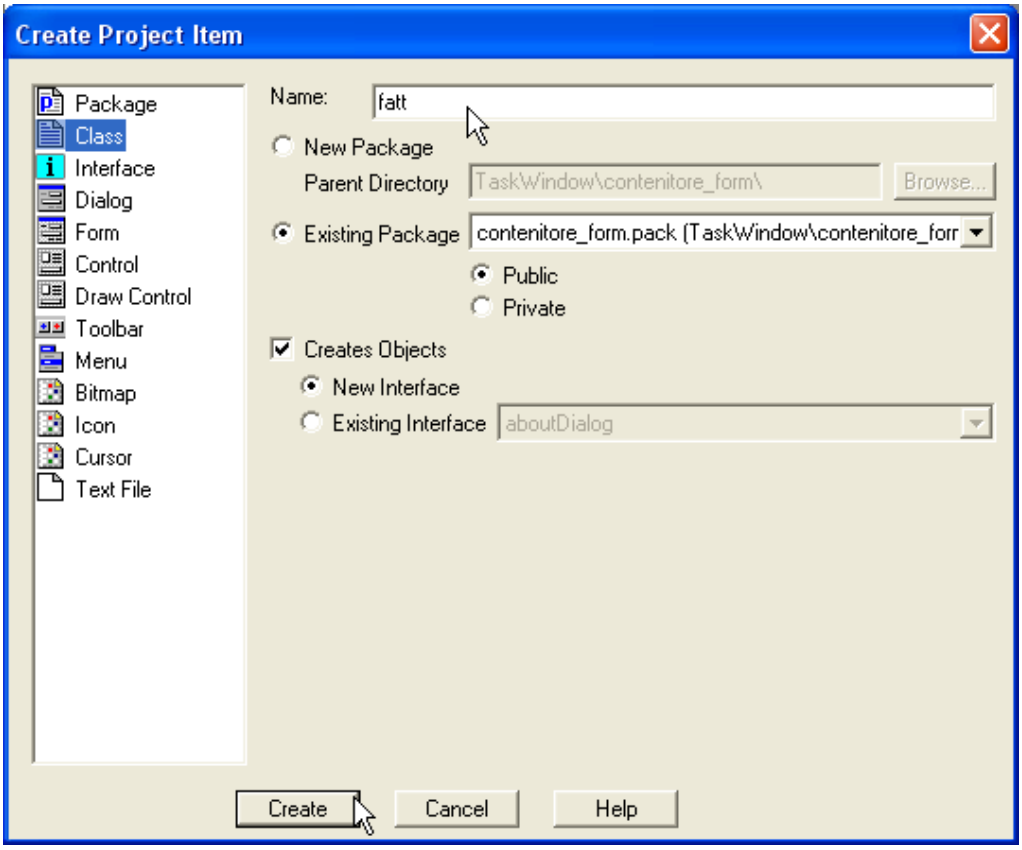

FIGURA 3.1.15. Creare una nuova classe

L'unico campo che bisogna completare è quello del nome della classe, scegliendo il pulsante Create si ottengono tre nuovi file riportati :

- fatt.pro nella figura 3.1.16 nella pagina successiva
- fatt.cl nella figura 3.1.17 a fronte
- fatt.i nella figura 3.1.18 nella pagina 52

In Visual Prolog il concetto di classe ricalca in qualche misura l'analogo concetto negli altri linguaggi orientati agli oggetti come Java, Pascal C++ e C#.Tralasciamo per ora il file fatt.i, che non dobbiamo modificare, ed osserviamo che il file fatt.cl indica i fatti, predicati, procedure, funzioni, termini che sono accessibili e possono essere usati anche all'esterno della classe stessa. Il file fatt.pro contiene il codice che implementa

```
fatt.pro (TaskWindow\contenitore_form\)
11:13    Insert    Indent
/*****************************************************

                Liceo Bodoni Saluzzo (CN)

*****************************************************

implement fatt
   open core

constants
   className = "TaskWindow/contenitore_form/fatt".
   classVersion = "".

clauses
   classInfo(className, classVersion).

end implement fatt
```

FIGURA 3.1.16. fatt.pro

```
fatt.cl (TaskWindow\contenitore_form\)
1:1    Insert    Indent
/*****************************************************

                Liceo Bodoni Saluzzo (CN)

*****************************************************
class fatt : fatt
   open core

predicates
   classInfo : core::classInfo.
   % @short Class information  predicate.
   % @detail This predicate represents information predicate of this class.
   % @end

end class fatt
```

FIGURA 3.1.17. fatt.cl

quanto dichiarato nel file fatt.cl e può contenere la definizione e l'implementazione di fatti, predicati, termini, procedure e funzioni che non sono accessibili all'esterno del file fatt.pro stesso. Il file fatt.cl deve essere modificato come indicato nella figura 3.1.19 nella pagina seguente

L'unica cosa aggiunta è la riga

FIGURA 3.1.18. fatt.i

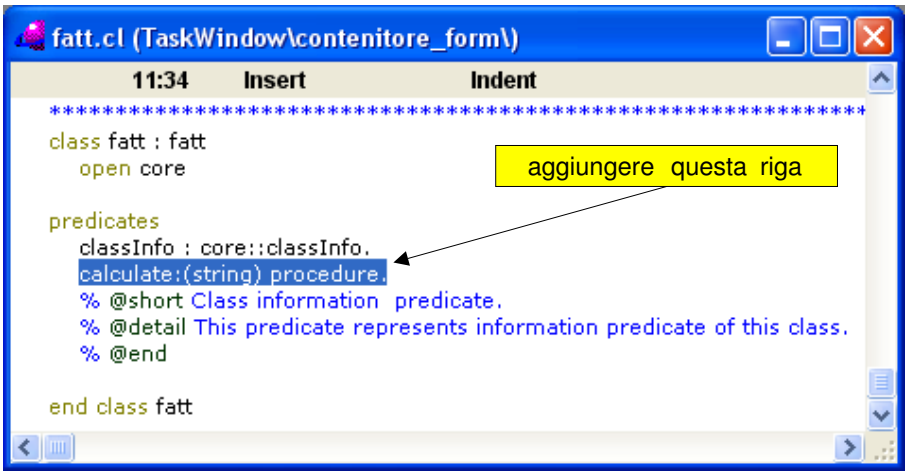

FIGURA 3.1.19. Modifica a fatt.cl

```
calculate:(string) procedure.
```

che introduce nella classe la procedura che servirà per agganciare i dati inseriti nella casella di testo Edit-ctl, tramite l'azione del pulsante fattoriale della form inserimento_dati, alla classe fatt appena introdotta. Per fare questo bisogna ritornare all'albero del progetto e scegliere il pulsante fattoriale nella form inserimento_dati e visualizzare la finestra di dialogo properties

Facendo doppio click come indicato in figura 3.1.20 nella pagina successiva viene automaticamente creata la procedura onFattorialeClick nel file inserimento_dati.pro come riportato in figura 3.1.21 a fronte

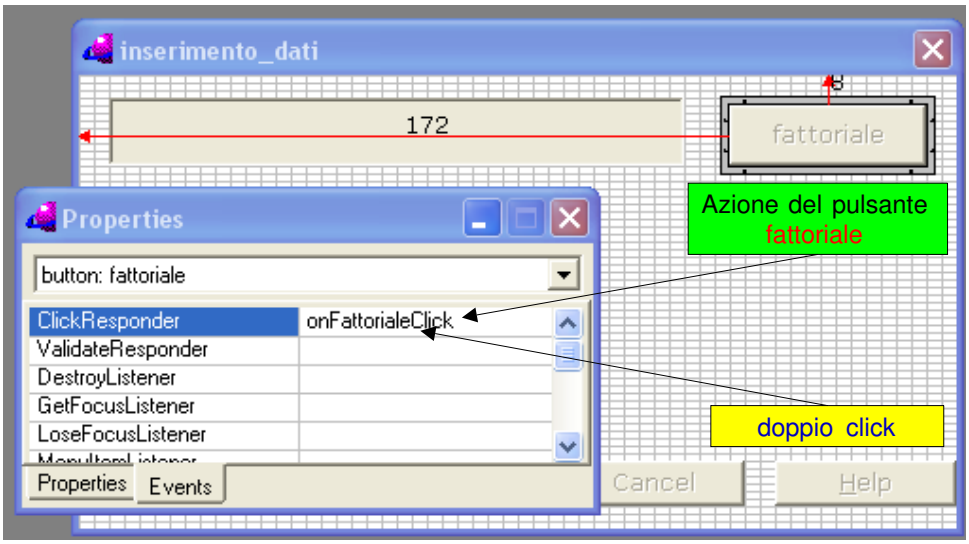

FIGURA 3.1.20. Azione del pulsante fattoriale

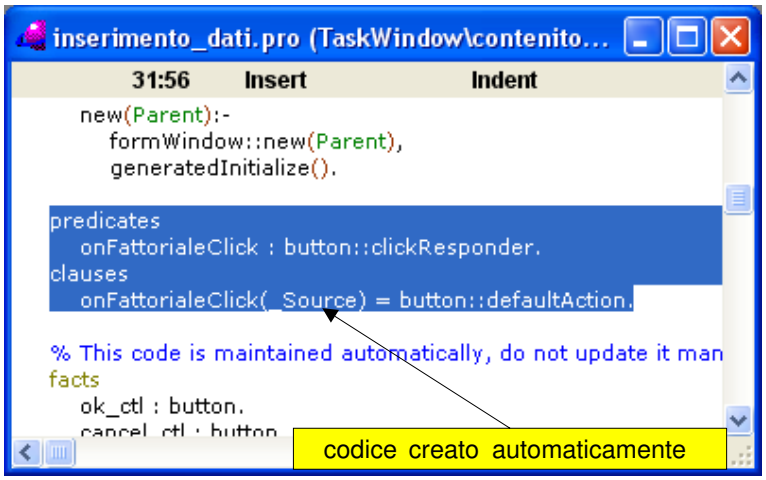

FIGURA 3.1.21. OnFattorialeClick

se lasciamo inalterato questa parte del codice quando premiamo il pulsante fattoriale non succede nulla, è qui che dobbiamo agganciare la classe fatt, pertanto il codice va modificato come indicato nella figura 3.1.22 nella pagina successiva

Una volta che abbiamo agganciato la classe fatt quando facciamo click sul pulsante fattoriale nella form inserimento dati viene richiamata la procedura la cui interfaccia[1] è:

```
calculate:(string) procedure.
```

[1]La dichiarazione di questa interfaccia si trova nel file fatt.cl

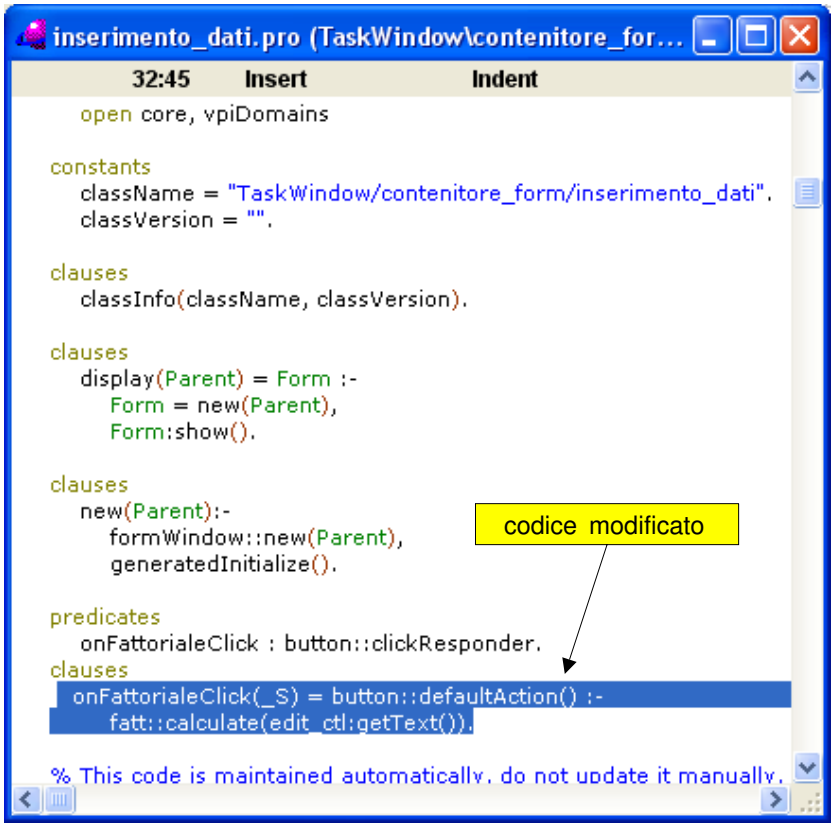

FIGURA 3.1.22. onFattorialeClic (modificato)

La sua implementazione si trova nel file fatt.pro come riportato in figura 3.1.23 a fronte

Analizziamo il codice riportato in questo file. La parte di codice che viene immediatamente utilizzato è quello relativa alla procedura calculate(X) che riportiamo qui sotto

```
calculate(X) :- N= toterm(X), fact(N, F), stdio::write(F, "\n").
```

il predicato calculate richiama innanzitutto N= toterm(X), toterm(X) ha il compito di prendere la stringa X e di restituire un numero e, grazie all'operatore =, N viene unificato con questo numero. Si tratta quindi una funzione il cui funzionamento è descritto nell'Help di Visual Studio come segue.

fatt.pro (TaskWindow\contenitore_f...

18:1 Insert Indent

```
implement fatt
    open core

constants
    className = "TaskWindow/contenitore_form/fatt"
    classVersion = "1.0".

class predicates
    fact:(integer, integer) procedure (i,o).
clauses
    classInfo(className, className, classVersion).
    fact(0, 1) :- !.
    fact(N, N*F) :- fact(N-1, F).
    calculate(X) :- N= toterm(X),
    fact(N, F), stdio::write(F, "\n").

end implement fatt
```

FIGURA 3.1.23. File fatt.pro, versione definitiva

```
::toTerm/1-> toTerm/1-> procedure (i).
```

Converts the string/binary representation of the specified term SrcTerm into representation corresponding to the domain of PrologTerm variable of the return value.

Description Call-template for this function is:

PrologTerm = toTerm (SrcTerm)

The compiler should be able to determine the domain for the returned value PrologTerm while compiling-time. Notice that binary version of toTerm predicate performs almost byte to byte conversion and only checking general compatibility of SrcTerm data with the domain required to the return value PrologTerm. The programmer is wholly responsible for providing binary data of SrcTerm that can be correctly converted to the term of the desired domain. The toTerm predicates are counterparts to predicates toBinary/1-> and toString/1->. When a Term (of some domain domainName) is converted into a binary or string representation SrcTerm (by toBinary/1-> or toString/1-> correspondingly), it can safely be stored in a file or sent over a network to another program. Later the corresponding toTerm/1-> function can convert the obtained string/binary value SrcTerm back to a Visual Prolog term PrologTerm. For correctness of the reverse conversion the domain of the clause variable PrologTerm should be adequate to the initial domain domainName.

See also tryToTerm.

Exceptions Compile time error if the compiler cannot determ the return value domain.

Run time errors are generated when the toTerm/1-> predicate cannot convert the string or binary into a term of the specified domain.

La parte incorniciata è quanto si trova nell'Help di Visual Prolog, essenzialmente dice che toTerm(SrcTerm) è da vedersi come una funzione che prende la stringa SrcTerm e la trasforma in un tipo di dato che è quello che il Prolog richiede nell'istruzione in cui toTerm è usata. Noi la usiamo in

```
calculate(X) :- N= toterm(X), fact(N, F), stdio::write(F, "\n").
```

quindi il suo risultato deve essere assegnato a N il quale deve essere usato in fact(N,F) ciè N deve essere un integer come è stato definito nella riga di codice di fatt.pro

```
fact:(integer, integer) procedure (i,o).
```

La procedura stdio::write(F, "\n") non fa altro che stampare sull'output standard[2] il contenuto di F.

[2]La finesta Messages

Vogliamo modificare il programma in modo che l'output non venga più stampato nella finestra Messages ma venga inserito in un apposito campo di testo. Apriamo la form Inserimento-dati, e aggiungiamo un nuovo campo di testo edit1_ctl come è indicato nella figura

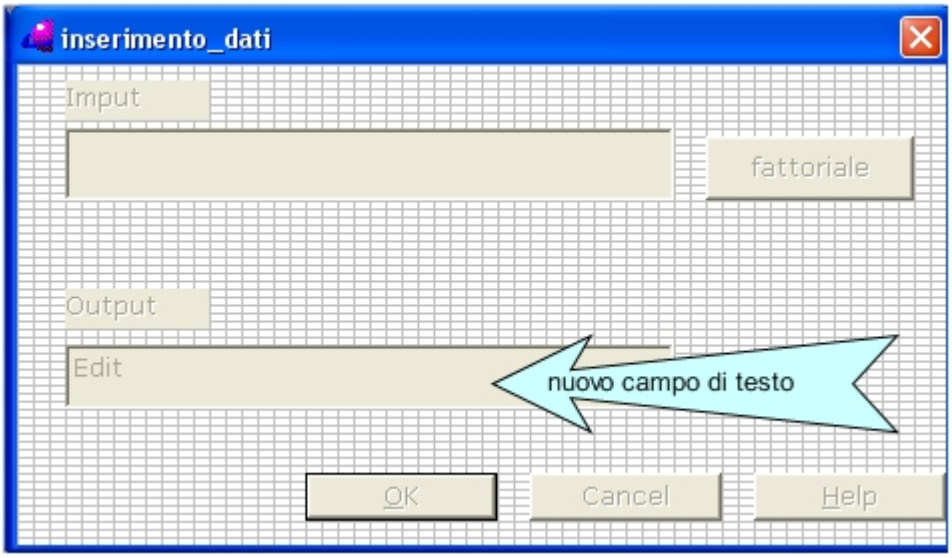

FIGURA 3.1.24. Form inserimento_dati con un nuovo campo di testo

Nella classe fatt bisogna modificare la procedura calculate nel seguente modo (vedi figura 3.1.25 nella pagina successiva)

Il predicato *calculate*, per avere successo, deve richiamare Y=toString(F) questa è un funzione che accetta in imput F (che deve essere già istanziato a un numero) e restituisce lo stesso numero trasformato in stringa. Ecco quanto scritto nell'help di Visual Prolog

```
::toString/1-> toString/1-> string procedure (i).
```

Converts the specified Term to the string representation.

Description Call-template for this function is:

```
String_Term = toString ( Term )
```

When a Term (of some domain domainName) is converted into a string, it can safely be stored in a file or sent over a network to another program. Later the obtained string value can be converted back to a Visual Prolog term, using toTerm/1-> function (the domain of the return value should be adequate to domainName) for the reverse conversion.

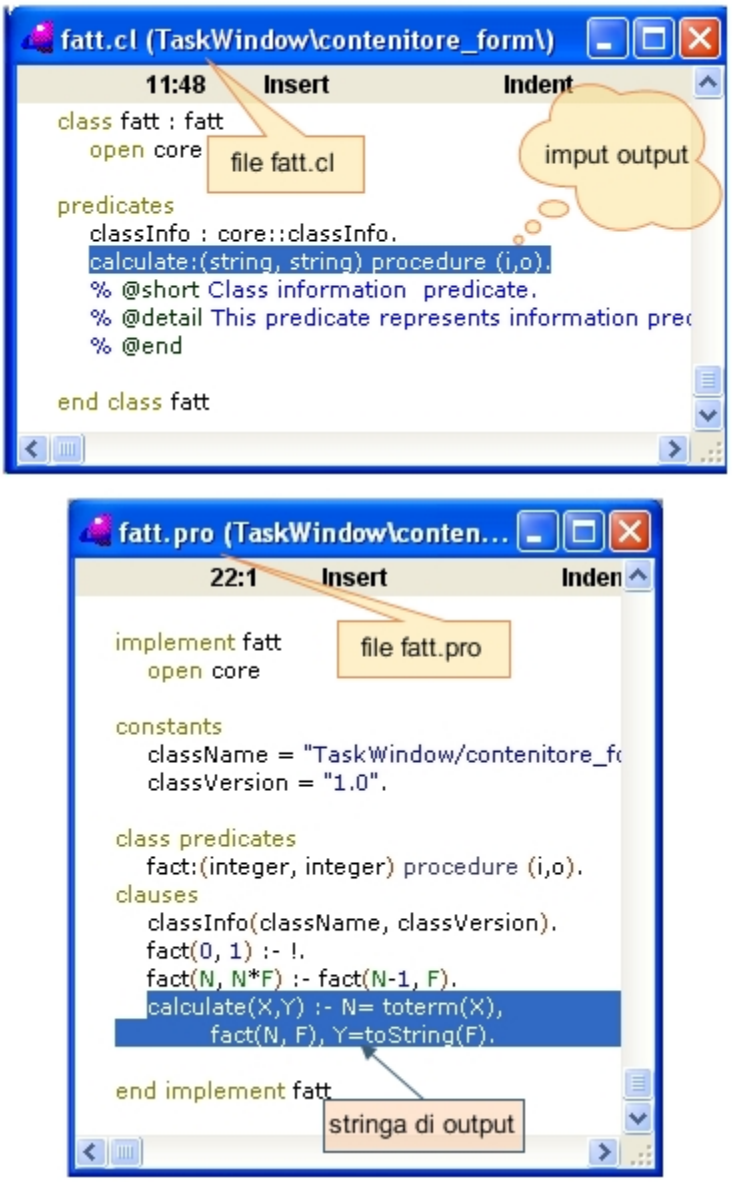

FIGURA 3.1.25. Modifiche alla procedura *calculate* della classe *fatt*

A questo punto bisogna mettere questa stringa nella nuova casella di testo. Per fare questo bisogna modificare il file inserimento_dati.pro come indicato nella figura 3.1.26 a fronte Ecco che cosa succede quando si esegue il programma (vedi figura 3.2.14 nella pagina 73)

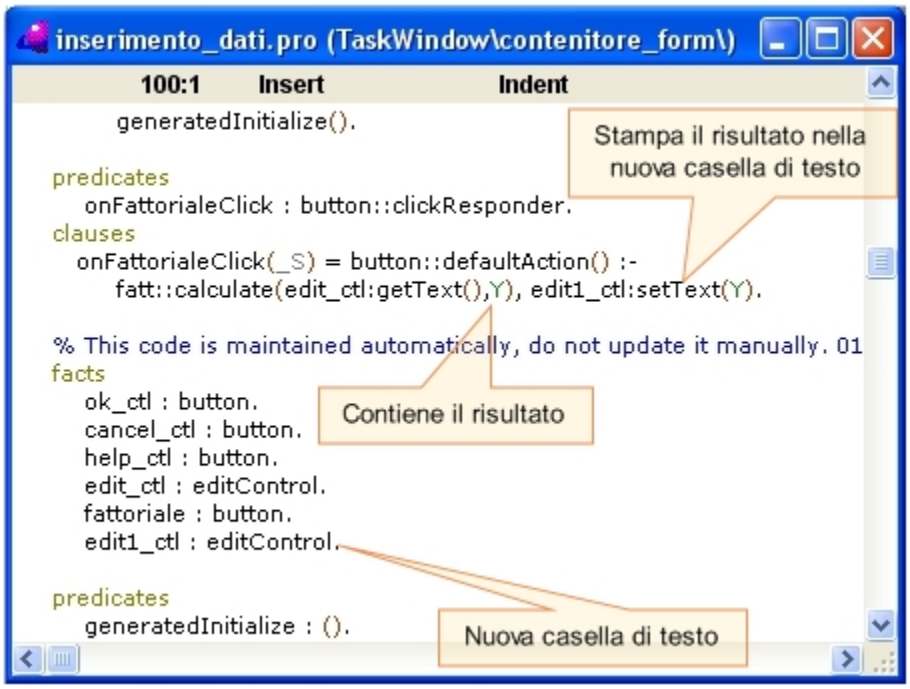

FIGURA 3.1.26. Medifica a *inserimento_dati.pro*

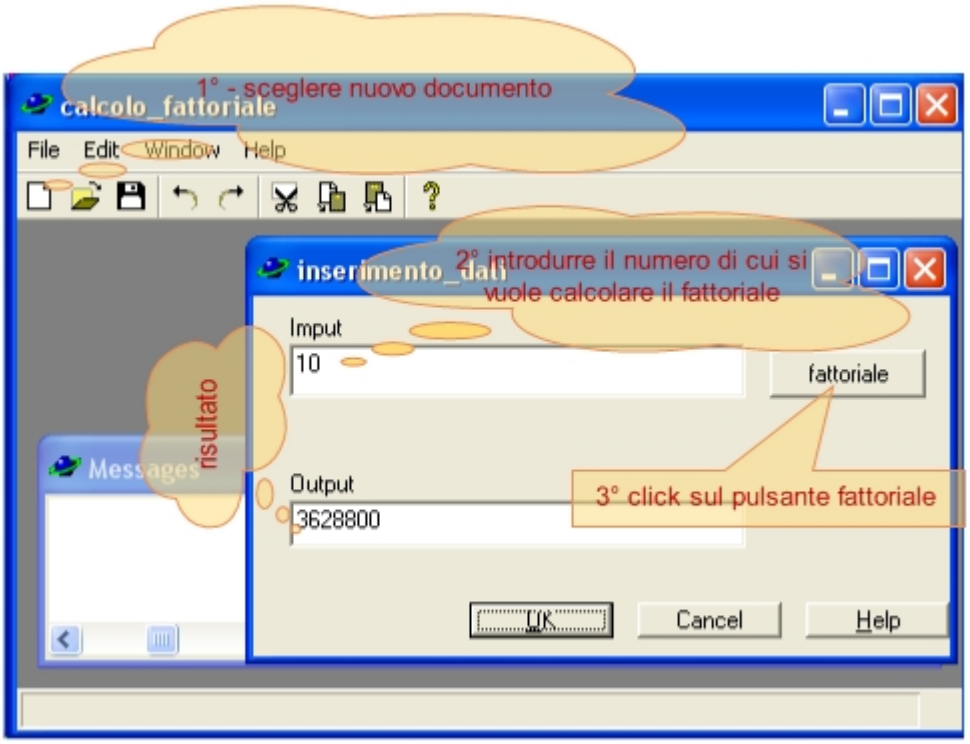

FIGURA 3.1.27. Esecuzione del programma

3.2. Il programma mappaBaseDati

Costruiamo un nuovo programma nel corso del quale impareremo ad usare delle procedure grafiche. Costruiamo un nuovo progetto eseguendo i soliti passi:

Primo passo: scegliete New dal menù Project (oppure utilizzare Ctrl+Shift+N), apparirà la finestra di dialogo riportata nella figura 3.2.1, che riempirete come indicato in detta figura.

Secondo passo: scegliete New in New Package... dal menù File ed infine scegliete Form che riempirete con i tre pulsanti riportati nella figura 3.2.2 nella pagina successiva. Dimensionate la form in modo che nella sua parte centrale ci possa stare la mappa.

Terzo passo: compilate ed eseguite il progetto.

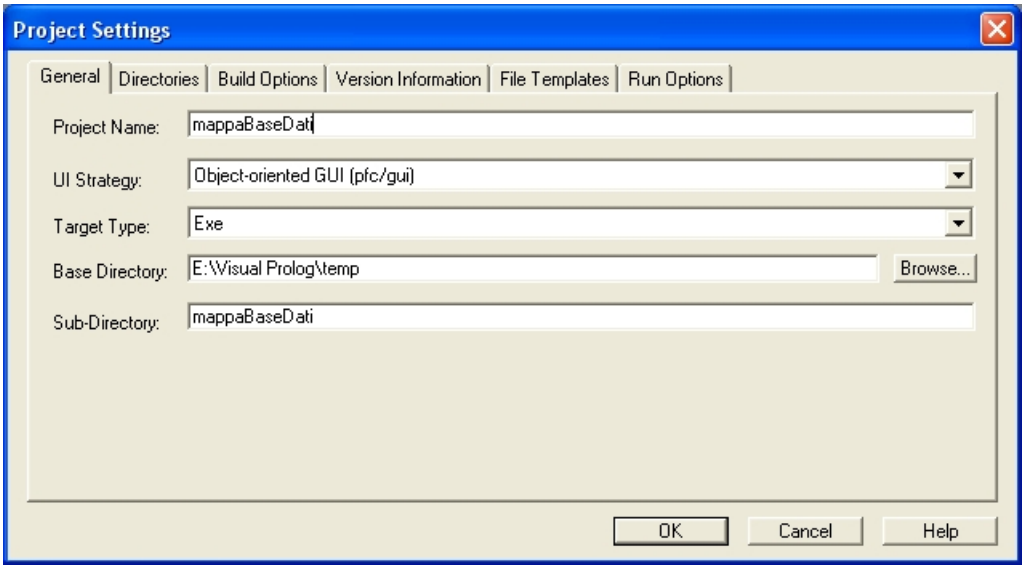

FIGURA 3.2.1. Progetto mappaBaseDati

La compilazione è necessaria perché durante questo processo vengono costruiti nuovi file necessari in seguito.

Come già nei programmi precedenti vogliamo che il comando New , sia dal menu File che dalla barra delle icone-comandi faccia apparire la nuova form. Apriamo l'albero del progetto, come è indicato in figura 3.2.3 nella pagina 63 e scegliamo TaskMenu.mnu.

Comparirà la finestra di dialogo riportata in figura 3.2.4 nella pagina 64 scegliete prima &File poi &New\F7 abbandonate la finestra e vi verrà chiesto di salvare il tutto. Ritornate sull'albero del progetto (figura 3.2.5 nella pagina 65) fate doppio click su TaskWindow.Win comparirà la finestra di dialogo riportata in figura 3.2.6 nella pagina 66.

Fate click su Code Expert e comparirà la finestra di dialogo riportata in figura 3.2.7 nella pagina 67 Scegliete prima Menu, poi TaskMenu, poi id_file ed infine fate

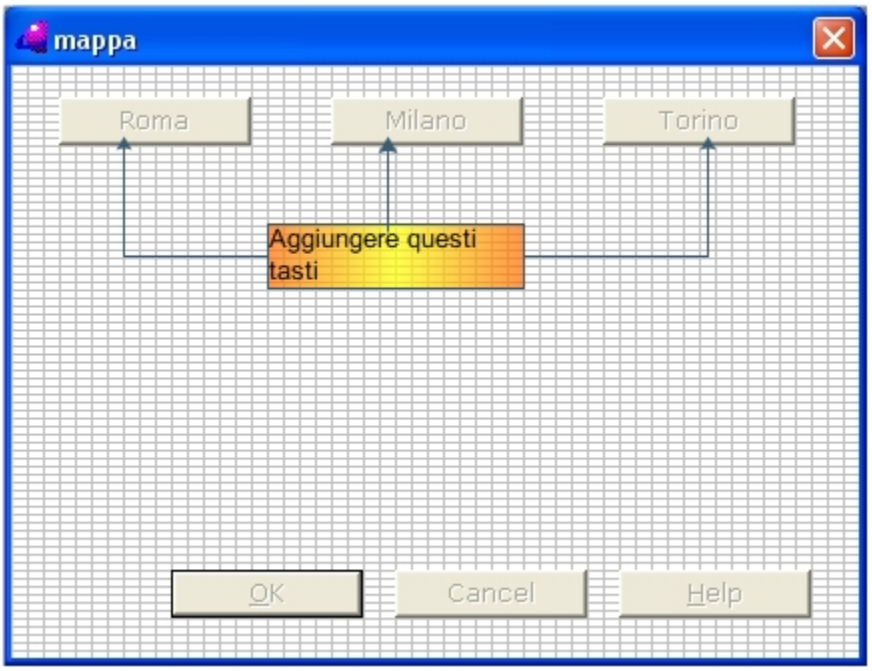

FIGURA 3.2.2. La form *mappa*

doppio click su id_file_new comparirà la finestra TaskWindow.pro riportata in figura 3.2.8 nella pagina 68, compilate ed eseguite , quando farete click si New comparirà la form mappa.

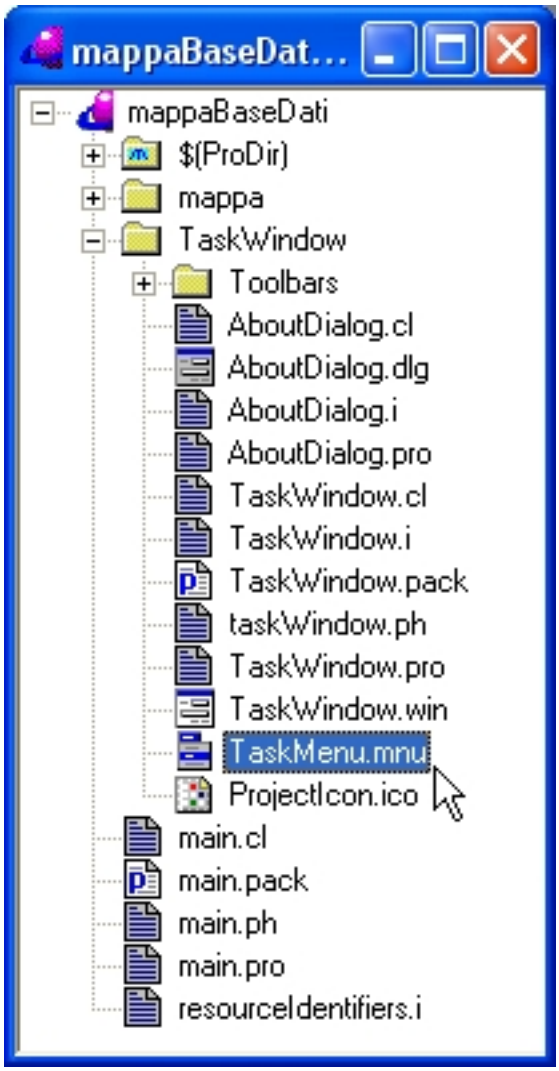

FIGURA 3.2.3. Albero del progetto

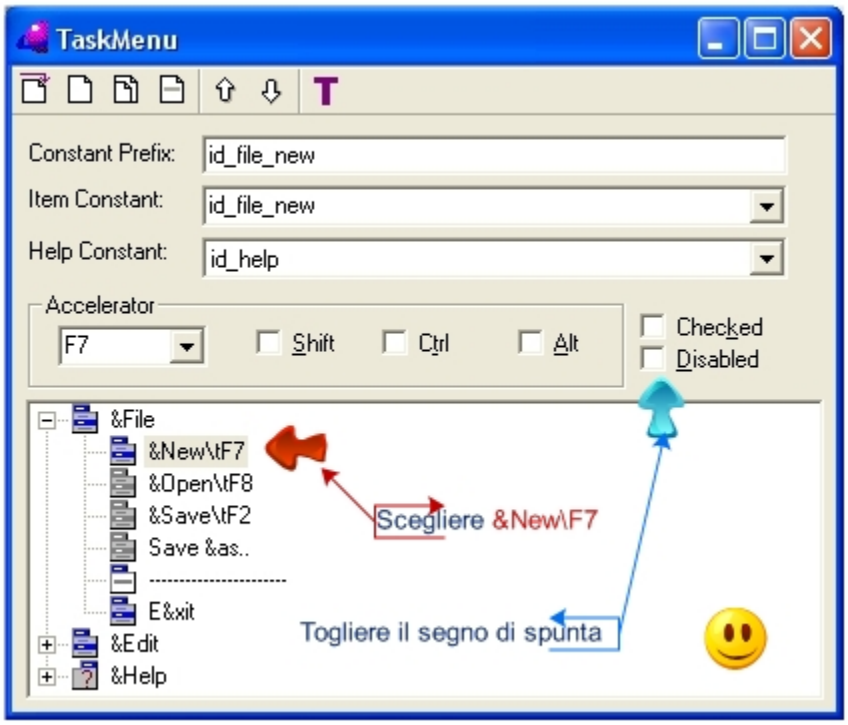

FIGURA 3.2.4. Abilitare File_New

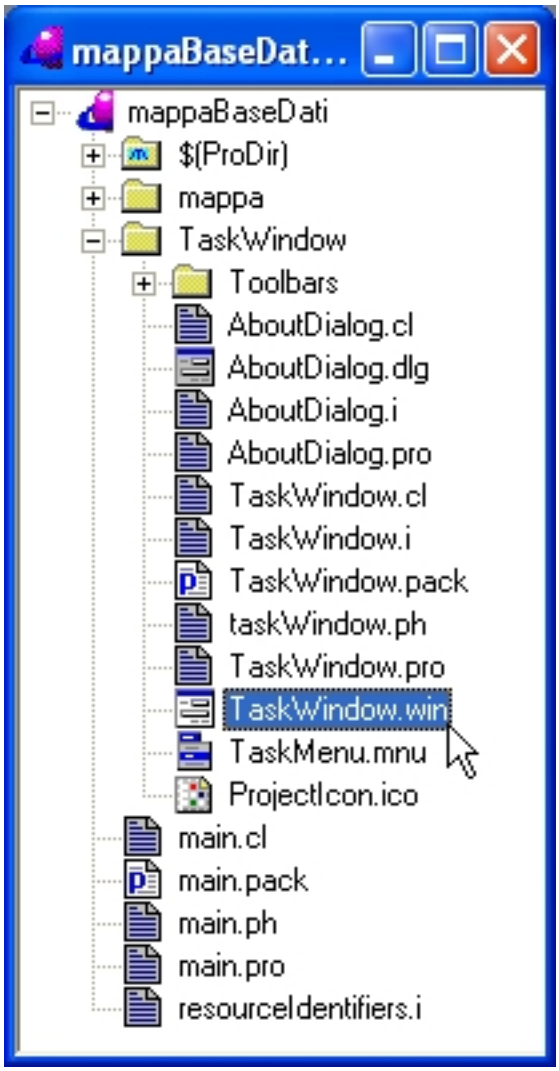

FIGURA 3.2.5. Albero del progetto, scegliere TaskWindow.Win

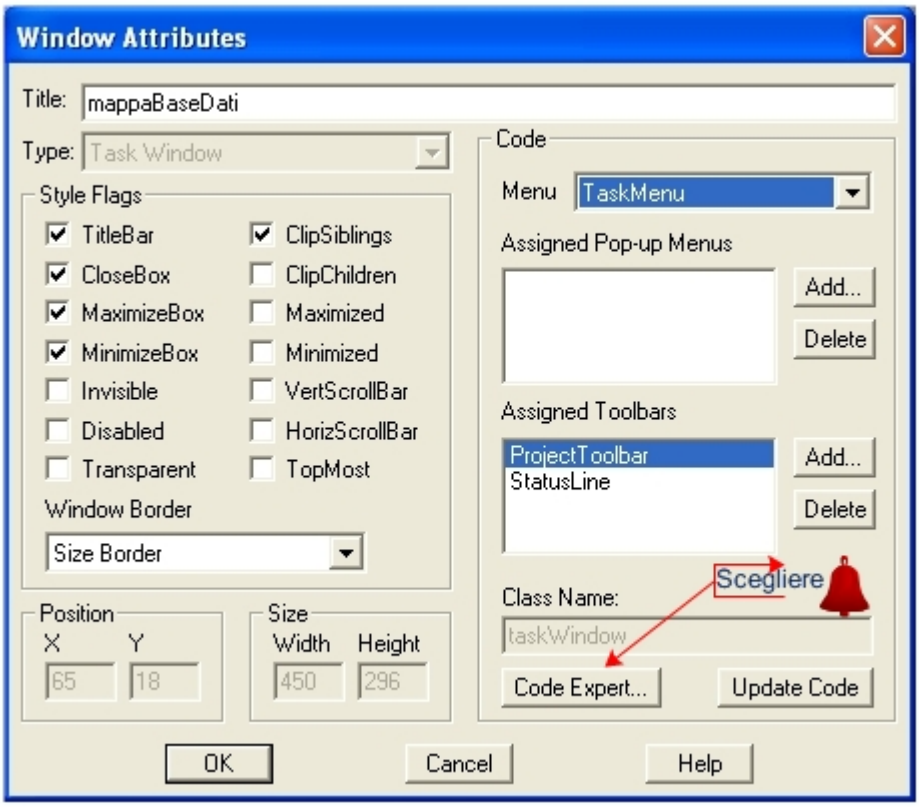

FIGURA 3.2.6. Code Expert di Windows Attribute

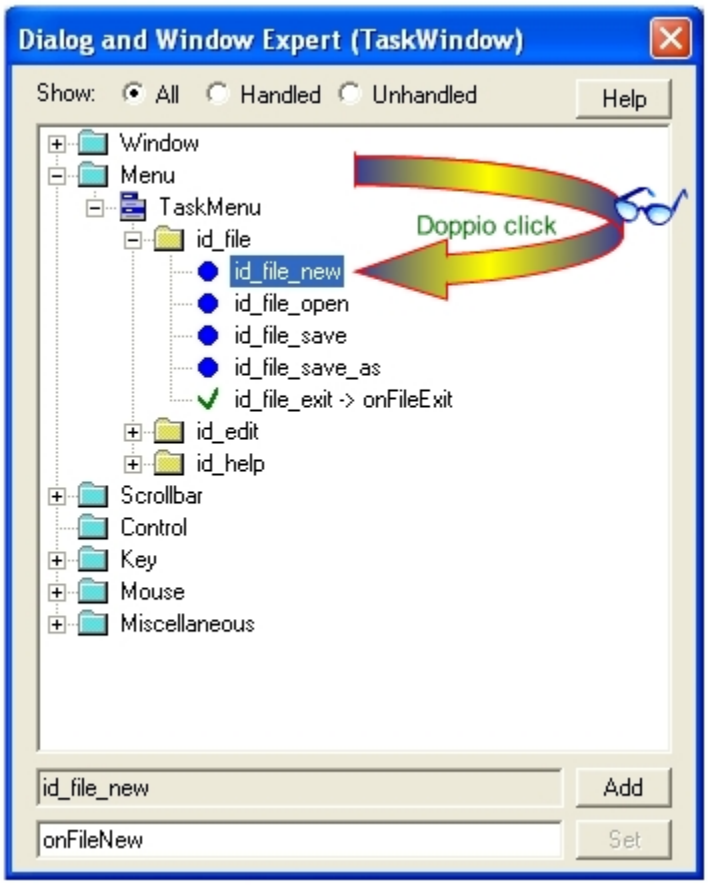

FIGURA 3.2.7. id_file_new (generare codice)

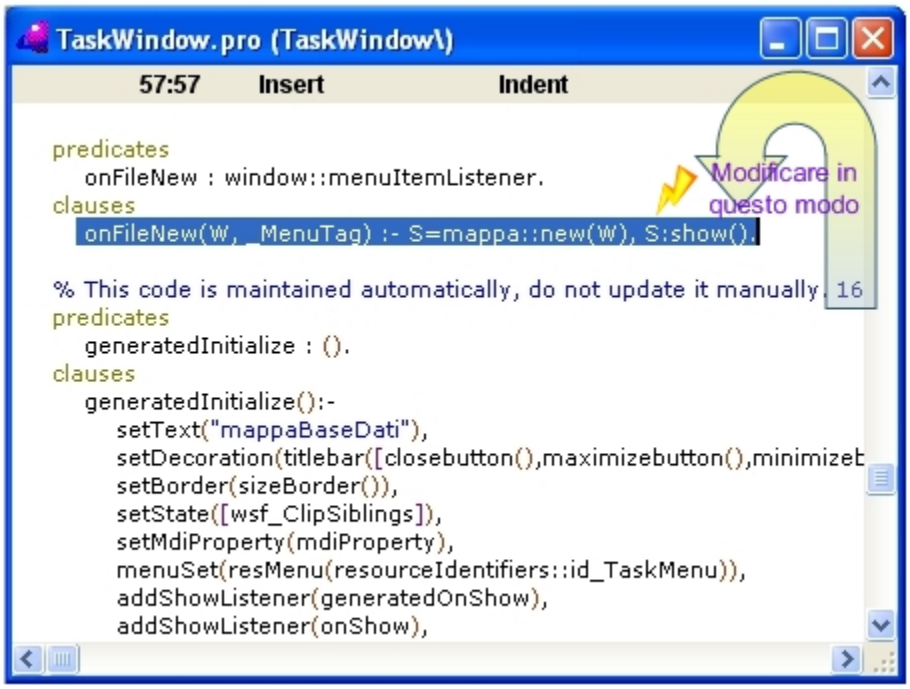

FIGURA 3.2.8. Modifica al codice prodotto da Code Expert

Per dare delle funzionalità al programma costruiamo una nuova classe. Nell'albero del progetto scegliamo la radice (vedi figura 3.2.9)

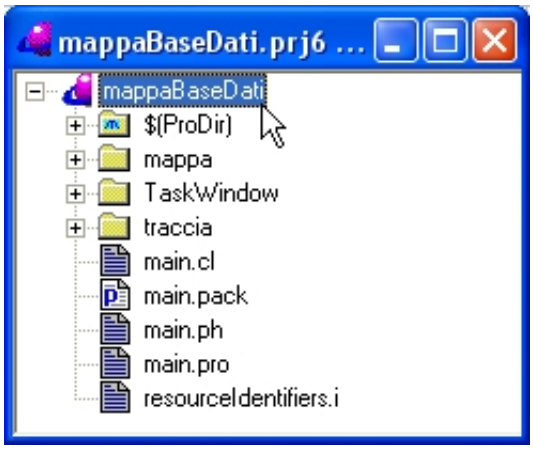

FIGURA 3.2.9. Scegliere la radice dell'albero del progetto

successivamente: facendo click nell'albero del progetto con il tasto destro, oppure scegliendo dalla barra delle icone, oppure scegliendo New in New Package dal menù file costruite una nuova classe come è indicato in figura 3.2.10 nella pagina successiva. Osservare come l'opzione Creates Objects non debba essere selezionata.

Creata la classe traccia compilare il tutto e successivamente modificare i file traccia.cl e traccia.pro, come indicato nelle figure 3.2.11 nella pagina seguente e 3.2.12 nella pagina 71

Infine ritornate sulla form di figura 3.2.2 nella pagina 62 contenente i quattro pulsanti: Roma, Milano, Torino, e per ciascuno di essi, nella casella di dialogo Properties scegliere Events (vedere in basso nella casella di dialogo) e in corrispondenza dell'evento ClickResponder scegliete OnRomaClick, fate doppio click su esso e verrete automaticamente riportati sul file mappa.pro dove Code Expert ha già predisposto lo scheletro della clausola OnRomaClick, che voi modificherete come indicato nella figura 3.2.13 nella pagina 72

Ripetete queste operazioni per gli altri pulsanti, infine compilate ed eseguite ed ecco il funzionamento del programma (vedi figura 3.2.14 nella pagina 73)

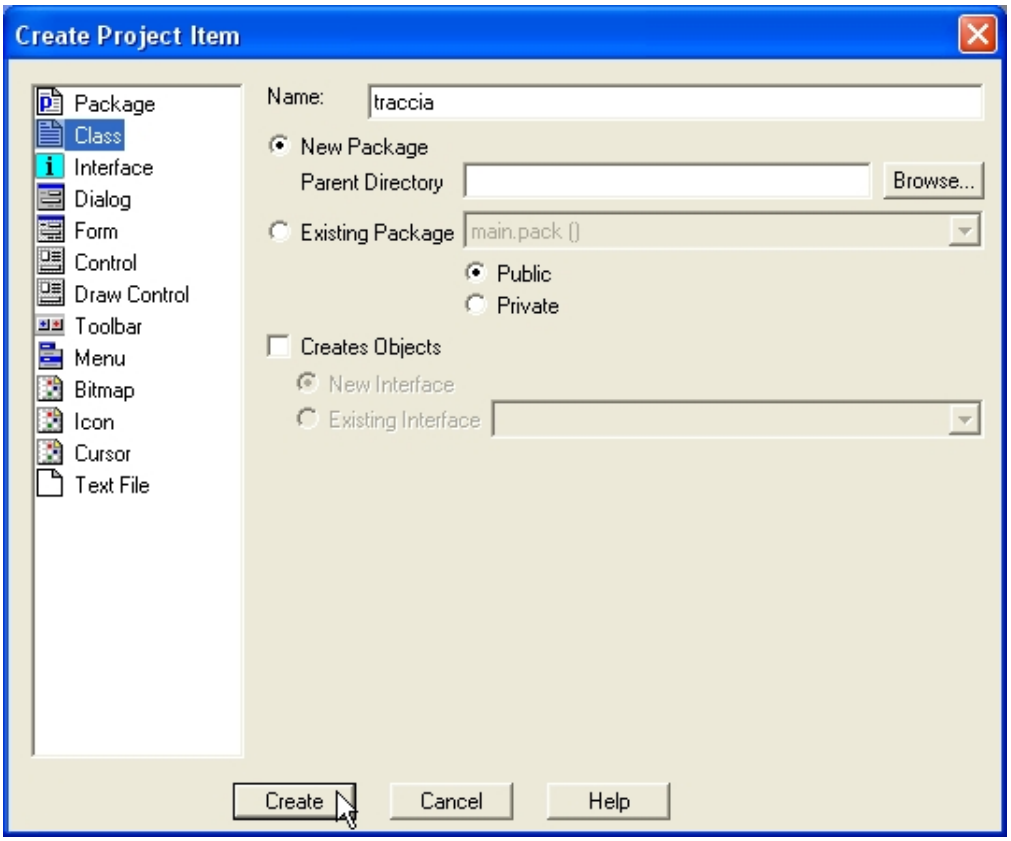

FIGURA 3.2.10. Creare la classe *traccia*

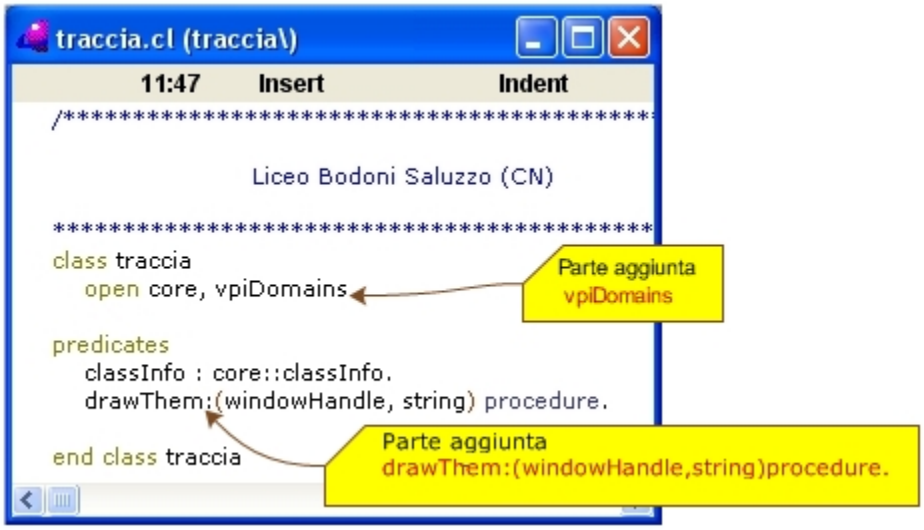

FIGURA 3.2.11. Traccia.cl

traccia.pro (traccia\)

32:1 Insert Indent

```
**********************************************************

implement traccia
    open core, vpiDomains, vpi

constants
    className = "traccia/traccia".
    classVersion = "".

class facts
    city:(string, pnt).

clauses
    classInfo(className, classVersion).

    city("Roma", pnt(30, 40)).
    city("Milano", pnt(100, 120)).
    city("Torino", pnt(100, 80)).
    city("Firenze", pnt(200, 100)).
    drawThem(Win, Name) :- B= brush(pat_solid, color_red),
    winSetBrush(Win, B),
    city(Name, P), !, P= pnt(X1, Y1),
    X2= X1+20, Y2= Y1+20,
    drawEllipse(Win, rct(X1, Y1, X2, Y2)).
    drawThem(_Win, _Name).

end implement traccia
```

FIGURA 3.2.12. Traccia.pro

mappa.pro (mappa\)

58:3 Insert Indent

```
predicates
   onRomaClick : button::clickResponder.
clauses
   onRomaClick(S) = button::defaultAction() :-
   Parent= S:getParent(),
   P= Parent:getVPIWindow(),
   traccia::drawThem(P, "Roma").

predicates
   onMilanoClick : button::clickResponder.
clauses
   onMilanoClick(S) = button::defaultAction() :-
   Parent= S:getParent(),
   P= Parent:getVPIWindow(),
   traccia::drawThem(P, "Milano").

predicates
   onTorinoClick : button::clickResponder.
clauses
   onTorinoClick(S) = button::defaultAction() :-
   Parent= S:getParent(),
   P= Parent:getVPIWindow(),
   traccia::drawThem(P, "Torino").

predicates
   onFirenzeClick : button::clickResponder.
clauses
   onFirenzeClick(S) = button::defaultAction() :-
   Parent= S:getParent(),
   P= Parent:getVPIWindow(),
   traccia::drawThem(P, "Firenze").

% This code is maintained automatically, do not update it ma
```

FIGURA 3.2.13. Codice delle procedure $on\overbrace{\cdots\cdots}^{bottone}Click$

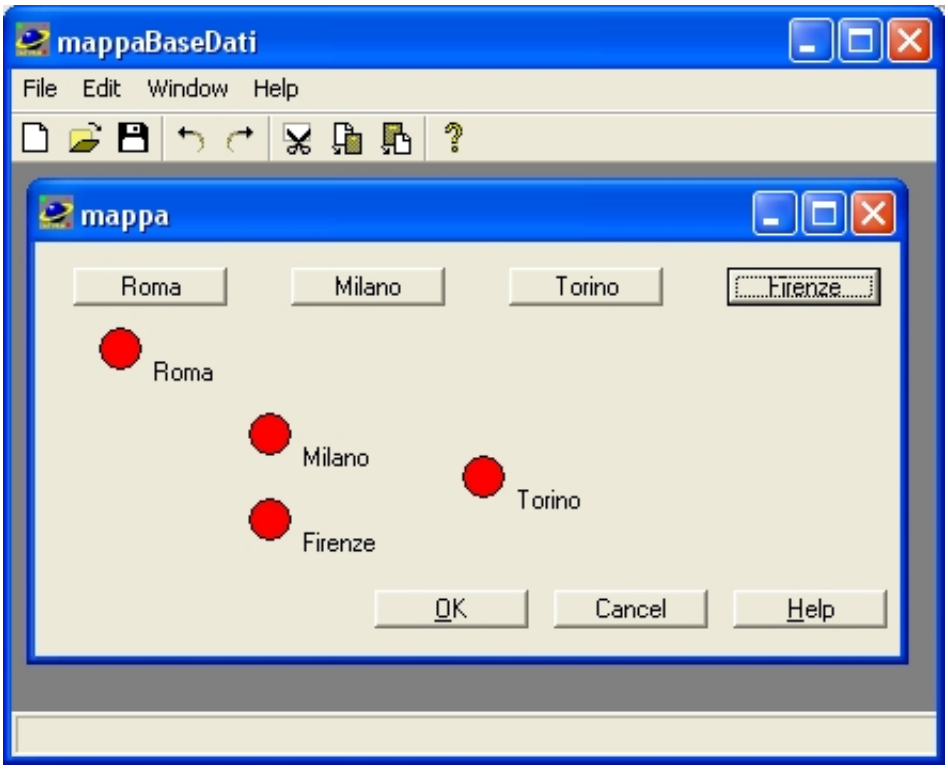

FIGURA 3.2.14. Esecuzione del programma *mappa*

Facciamo alcune osservazioni sul codice scritto in questa ultima parte. Innanzitutto osserviamo come nel costruire la classe traccia gli unici file, dei quattro generati dalla procedura automatica di costruzione di una classe, solo due, quello con estensione .cl e quello con estensione .pro, devono essere modificati manualmente. In entrambi i file vi è la sezione ***open*** che è una delle parti sulle quali bisogna intervenire manualmente. In questa sezione vengono indicate le classi che verranno utilizzate nel file in questione.

La classe ***core*** viene messa automaticamente, è la classe fondamentale contiene i vari tipi di dati standard e altre cose, di essa non ci si deve preoccupare viene gestita automaticamente.

La classe ***vpiDomains***, deve essere aggiunta manualmente in entrambi i file . Ecco che cosa si dice di essa nella guida di Visual Prolog

Class vpiDomains
Package: vpi.vpiDomains
Class vpiDomains contains domains for GUI objects: windows, dialogs, menus, controls, pictures and so on.

In particolare questa classe contiene ***brush*** che indica il modo con cui vengono disegnati rettangoli, ellissi, cerchi, etc...:

vpiDomains::brush
brush = brush(patStyle PatStyle, color Color).
Specifies a mode of filling rectangles, ovals, polygons, etc.

patStyle indica lo stile dello riempimento delle varie figura, ecco che cosa c'è scritto nella guida:

vpiDomains::patStyle

patStyle = integer.

Brush style. Specifies a fill mode used when an object is drawn.
Predefined constants are:

- ***pat_Hollow*** Hollow brush (nessun tratteggio)
- ***pat_Solid*** Solid brush (colore uniforme)
- ***pat_Horz*** Horizontal hatch (tratteggio orizzontale)
- ***pat_Vert*** Vertical hatch (tratteggio verticale)
- ***pat_FDiag*** 45-degree downward hatch (tratteggio in diagonale)
- ***pat_BDiag*** 45-degree upward hatch (tratteggio in diagnale)
- ***pat_Cross*** Horizontal and vertical cross-hatch (tratteggio incrociato, verticale orizzontale)
- ***pat_DiagCross*** 45-degree cross hatch (tratteggio incrociato in diagonale)

patStyle constants Brush style. Specifies a fill mode used when an object is drawn.

- ***pat_Hollow = 1.***
- ***pat_Solid = 2.***
- ***pat_Horz = 3.***
- ***pat_Vert = 4.***
- ***pat_FDiag = 5.***
- ***pat_BDiag = 6.***
- ***pat_Cross = 7.***
- ***pat_DiagCross = 8.***

Color indica il colore utilizzato nello riempimento, ecco che cosa è riportato nella guida

vpiDomains::color, vpiDomains::colorList

color = integer. colorList = color*.

Specifies three bytes of RGB color.
Some colors can be specified using color_*** predefined constants.

I codici dei colori sono riportati nelle figure 3.2.15 nella pagina successiva e 3.2.16 nella pagina 77

La classe vpiDomains contiene anche ***pnt***, ecco che cosa si dice sulla guida:

vpiDomains::pnt, vpiDomains::pntlist

pnt = pnt(integer X, integer Y).

pntlist = pnt*.

Point coordinates.
pnt(0,0) is the upper left corner.

Sempre la stessa classe contiene anche rct il quale è così illustrato sulla guida:

```
color_Red = 0x000000FF.
color_Green = 0x0000FF00.
color_Blue = 0x00FF0000.
color_Cyan = 0x00FFFF00.
color_Magenta = 0x00FF00FF.
color_Yellow = 0x0000FFFF.
color_Black = 0x00000000.
color_DkGray = 0x00404040.
color_Gray = 0x00808080.
color_LtGray = 0x00C0C0C0.
color_White = 0x00FFFFFF.
color_AliceBlue = 0x00FFF8F0.
color_AntiqueWhite = 0x00D7EBFA.
color_Aqua = 0x00FFFF00.
color_Aquamarine = 0x00D4FF7F.
color_Azure = 0x00FFFFF0.
color_Beige = 0x00DCF5F5.
color_Bisque = 0x00C4E4FF.
color_BlanchedAlmond = 0x00CDEBFF.
color_BlueViolet = 0x00E22B8A.
color_Brown = 0x002A2AA5.
color_BurlyWood = 0x0087B8DE.
color_CadetBlue = 0x00A09E5F.
color_Chartreuse = 0x0000FF7F.
color_Chocolate = 0x001E69D2.
color_Coral = 0x00507FFF.
color_CornFlowerBlue = 0x00ED9564.
color_Cornsilk = 0x00DCF8FF.
color_Crimson = 0x003C14DC.
color_DarkBlue = 0x008B0000.
color_DarkCyan = 0x008B8B00.
color_DarkGoldenRod = 0x000B86B8.
color_DarkGray = 0x00A9A9A9.
color_DarkGreen = 0x00006400.
color_DarkKhaki = 0x006BB7BD.
color_DarkMagenta = 0x008B008B.
color_DarkOliveGreen = 0x002F6B55.
color_DarkOrange = 0x00008CFF.
color_DarkOrchid = 0x00CC3299.
color_DarkRed = 0x0000008B.
color_DarkSalmon = 0x007A96E9.
color_DarkSeaGreen = 0x008FBC8F.
color_DarkSlateBlue = 0x008B3D48.
color_DarkSlateGray = 0x004F4F2F.
color_DarkTurquoise = 0x00D1CE00.
color_DarkViolet = 0x00D30094.
color_DeepPink = 0x009314FF.
color_DeepSkyBlue = 0x00FFBF00.
color_DimGray = 0x00696969.
color_DodgerBlue = 0x00FF901E.
color_FireBrick = 0x002222B2.
color_FloraWhite = 0x00F0FAFF.
color_ForestGreen = 0x00228B22.
color_Gainsboro = 0x00DCDCDC.
color_GhostWhite = 0x00FFF8F8.
color_Gold = 0x0000D7FF.
color_GoldenRod = 0x0020A5DA.
color_GreenYellow = 0x002FFFAD.
color_HoneyDew = 0x00F0FFF0.
color_HotPink = 0x00B469FF.
color_IndianRed = 0x005C5CCD.
color_Indigo = 0x0082004B.
color_Ivory = 0x00F0FFFF.
color_Khaki = 0x008CE6F0.
color_Lavender = 0x00FAE6E6.
color_LavenderBlush = 0x00F5F0FF.
color_LawnGreen = 0x0000FC7C.
color_LemonChiffon = 0x00CDFAFF.
color_LightBlue = 0x00E6D8AD.
color_LightCoral = 0x008080F0.
color_LightCyan = 0x00FFFFE0.
color_LightGoldenRodYellow = 0x00D2FAFA.
color_LightGray = 0x00D3D3D3.
color_LightGreen = 0x0090EE90.
color_LightPink = 0x00C1B6FF.
color_LightSalmon = 0x007AA0FF.
color_LightSeaGreen = 0x00AAB220.
color_LightSkyBlue = 0x00FACE87.
color_LightSlateBlue = 0x00FF7084.
color_LightSteelBlue = 0x00DEC4B0.
```

FIGURA 3.2.15. Codici dei colori - prima parte

vpiDomains::rct, vpiDomains::rct_list

rct = rct(integer Left, integer Top, integer Right, integer Bottom).

rct_list = rct*.

Coordinates of a rectangle.

```
color_LightYellow = 0x00E0FFFF.
color_Lime = 0x0000FF00.
color_LimeGreen = 0x0032CD32.
color_Linen = 0x00E6F0FA.
color_Maroon = 0x00000080.
color_MediumAquaMarine = 0x00AACD66.
color_MediumBlue = 0x00CD0000.
color_MediumGreen = 0x00008000.
color_MediumOrchid = 0x00D355BA.
color_MediumPurple = 0x00D87093.
color_MediumSeaGreen = 0x0071B33C.
color_MediumSlateBlue = 0x00EE687B.
color_MediumSpringGreen = 0x009AFA00.
color_MediumTurquoise = 0x00CCD148.
color_MediumVioletRed = 0x008515C7.
color_MidnightBlue = 0x00701919.
color_MintCream = 0x00FAFFF5.
color_MistyRose = 0x00E1E4FF.
color_Moccasin = 0x00B5E4FF.
color_NavajoWhite = 0x00ADDEFF.
color_Navy = 0x00800000.
color_OldLace = 0x00E6F5FD.
color_Olive = 0x00008080.
color_OliveDrab = 0x00238E6B.
color_Orange = 0x0000A5FF.
color_OrangeRed = 0x000045FF.
color_Orchid = 0x00D670DA.
color_PaleGoldenRod = 0x00AAE8EE.
color_PaleGreen = 0x0098FB98.
color_PaleTurquoise = 0x00EEEEAF.
color_PaleVioletRed = 0x009370D8.
color_PapayaWhip = 0x00D5EFFF.
color_PeachPuff = 0x00B9DAFF.
color_Peru = 0x003F85CD.
```

```
color_Pink = 0x00CBC0FF.
color_Plum = 0x00DDA0DD.
color_PowderBlue = 0x00E6E0B0.
color_Purple = 0x00800080.
color_RosyBrown = 0x008F8FBC.
color_RoyalBlue = 0x00E16941.
color_SaddleBrown = 0x0013458B.
color_Salmon = 0x007280FA.
color_SandyBrown = 0x0060A4F4.
color_SeaGreen = 0x00578B2E.
color_SeaShell = 0x00EEF5FF.
color_Sienna = 0x002D52A0.
color_Silver = 0x00C0C0C0.
color_SkyBlue = 0x00EBCE87.
color_SlateBlue = 0x00CD5A6A.
color_Snow = 0x00FAFAFF.
color_SpringGreen = 0x007AFF00.
color_SteelBlue = 0x00B48246.
color_Tan = 0x008CB4D2.
color_Teal = 0x00808000.
color_Thistle = 0x00D8BFD8.
color_Tomato = 0x004763FF.
color_Turquoise = 0x00D0E040.
color_Violet = 0x00EE82EE.
color_VioletRed = 0x009020D0.
color_Wheat = 0x00B3DEF5.
color_WhiteSmoke = 0x00F5F5F5.
color_YellowGreen = 0x0032CD9A.
color_Gray20 = 0x00202020.
color_Gray40 = 0x00404040.
color_Gray60 = 0x00606060.
color_Gray80 = 0x00808080.
color_GrayA0 = 0x00A0A0A0.
color_GrayC0 = 0x00C0C0C0.
color_GrayE0 = 0x00E0E0E0.
```

FIGURA 3.2.16. Codici dei colori - seconda parte

Nel file traccia.cl abbiamo dovuto aggiungere nella sezione open, vpiDomains perché successivamente dobbiamo usare **`windowHandle`** nella procedura drawThem, ecco che cosa si dice di windowHandle nella guida di Visual Prolog

```
vpiDomains::windowHandle
windowHandle = hwnd.
Handle to a window.
```

WindowHandle dà un riferimento alla finestra nella quale operare.

Continuiamo ad analizzare il file *traccia.cl*, dopo open incontriamo la sezione predicates ed in questa sezione vengono dichiarati i predicati della classe traccia che possono essere usati in tutti i file del progetto. La classe traccia è registrata nel progetto ed è quindi

conosciuta da tutti i file del progetto, i predicati dichiarati nel file *traccia.cl* possono essere usati ovunque nel progetto, gli altri predicati della classe traccia possono essere usati solo nel file *traccia.pro.*

Il predicato **`classInfo : core::classInfo.`** non fa altro che richiamare l'analogo predicato della classe *core* , non aggiunge nessuna nuova potenzialità.

Il predicato **`drawThem:(windowHandle, string) procedure.`** verrà usato dal file *mappa.pro* per gestire la pressione dei bottoni di quel form disegnando la posizione delle città in questione sul form stesso.

Nel file traccia.pro vi è la sezione

```
class facts

    city:(string, pnt).
```

I fatti sono delle clausole che non contengono variabili, infatti nella sezione clauses abbiamo i seguenti fatti:

```
city("Roma", pnt(30, 40)).

city("Firenze", pnt(100, 120)).

city("Milano", pnt(100, 80)).

city("Torino", pnt(200, 100)).
```

i quali costituiscono una base di dati.

Sempre nel file *traccia.pro* abbiamo usato due procedure interessanti *drawEllipse* e *drawText*, si trovano nella classe **`vpi`**, ecco cosa riporta la guida a questo proposito:

vpi::drawEllipse/2
`drawEllipse : ( vpiDomains::windowHandle WindowHandle, vpiDomains::rct BoundingRectangle) procedure (i,i)`
language c.
Draws an oval circle (ellipse).
Description The center of the ellipse is the center of the bounding rectangle BoundingRectangle. The ellipse is drawn by using the current window pen, and its interior is filled by using the current window brush. WindowHandle should be one of the drawable types. BoundingRectangle is the bounding rectangle of an ellipse.
Exceptions vpi::vpi_winBadHandle/3 vpi::vpi_winBadType/3 vpi::vpi_checkDraw/3 Drawing is forbidden while vpiDomains::e_EraseBackGround handling.

vpi::drawText/4

drawText : (vpiDomains::windowHandle WindowHandle, integer XStart, integer YStart, string StringToDraw) procedure (i,i,i,i).

Draws a whole contents of the StringToDraw string starting at the position given by the XStart and YStart coordinates.

Description WindowHandle should specify one of the drawable types. XStart - horizontal, YStart - vertical coordinates of the start position.

Exceptions vpi::vpi_winBadHandle/3 vpi::vpi_winBadType/3 vpi::vpi_checkDraw/3 Drawing is forbidden while vpiDomains::e_EraseBackGround handling.

3.3. Il programma SoluzioniMultiple

Costruite un nuovo progetto come è indicato nella figura 3.3.1 , successivamente andate sull'albero del progetto.

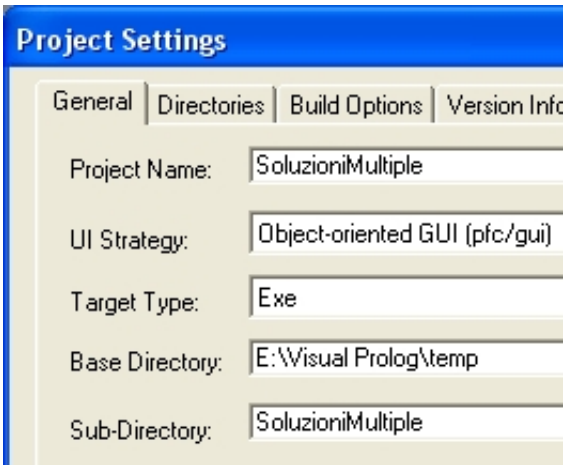

FIGURA 3.3.1. Inizio del nuovo progetto *SoluzioniMultiple*

Sull'albero del progetto selezionate il nodo SoluzioniMultiple e fate click con il tasto destro, nel pop-up selezionate New in New Package (vedi figura 3.3.2) .

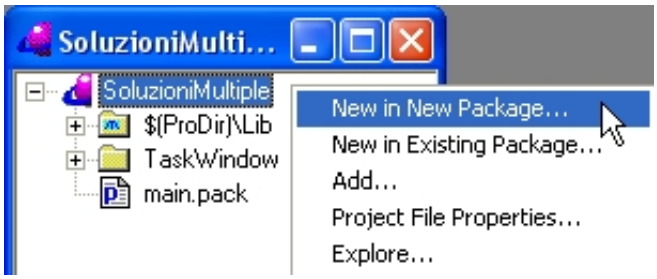

FIGURA 3.3.2. Costruzione del package *disegnatore* 1

Come conseguenza della vostra scelta apparirà la seguente finestra di dialogo (vedi figura 3.3.3 nella pagina successiva)

Dopo aver compilato la finestra di dialogo come indicato in figura 3.3.3 a fronte scegliete Create ed il package è costruito, lo potete constatare andando nell'albero del progetto. Aggiungiamo una form e la mettiamo nel package appena costruito. Andate nell'albero del progetto scegliete il nodo disegnatore , fate click con il tasto destro e nel pop-up scegliete New in Existing Package (vedere figura 3.3.4 nella pagina successiva).

Nella casella di dialogo che compare scegliete form e compilate i campi come è indicato nella figura 3.3.5 nella pagina 82.

Dopo aver fatto le scelte indicate nella figura citata precedentemente scegliete Create , infine compilate il progetto.

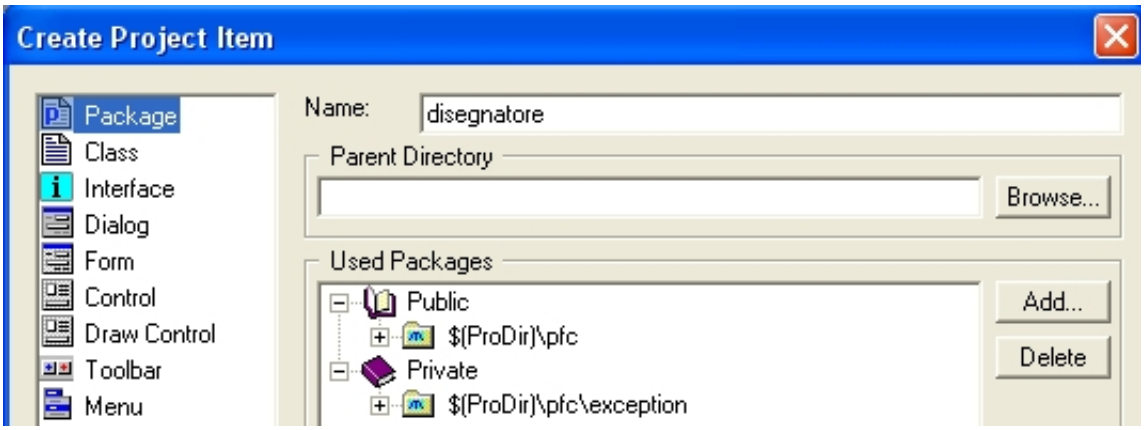

FIGURA 3.3.3. Costruzione del package *disegnatore* 2

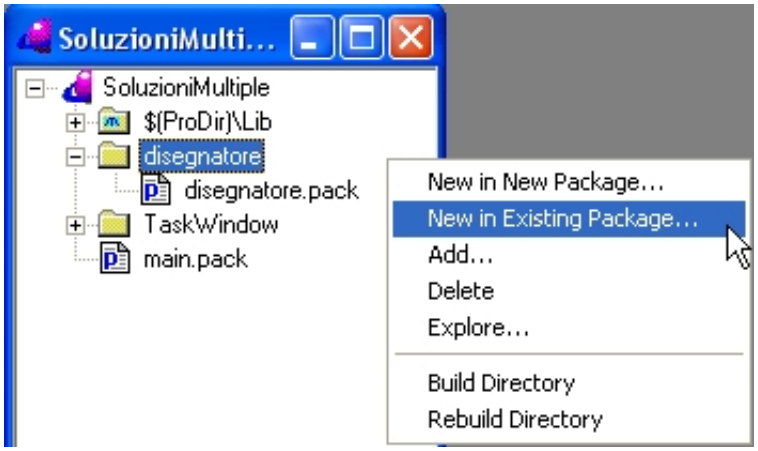

FIGURA 3.3.4. Inserire la form mappa 1

Abilitiamo File-New e colleghiamo ad esso la creazione della form mappa. Nell'albero del progetto apriamo TaskWindow e scegliamo TaskMenu.mu, come è indicato in figura 3.3.6 nella pagina successiva.

Facendo poi click con il tasto destro fate apparire il pop-up nel quale scegliete Edit, comparirà la finestra di dialogo della figura 3.3.7 nella pagina 83. In questa finestra scegliete prima File e poi New e togliete il segno di spunta dalla casella Disabled.

A questo punto ritornate sull'albero del progetto,nel nodo TaskWindow scegliete TaskWindow.win, vedi figura 3.3.8 nella pagina 83, agendo sul tasto destro fate apparire il pop-up nel quale scegliete CodeExspert, vi apparirà la finestra di dialogo della figura 3.3.9 nella pagina 84.

Scegliete Menu-TaskMenu-id_file-id_file_new, facendo click con il tasto destro fate apparire il pop-up nel quale sceglierete AddEvent.

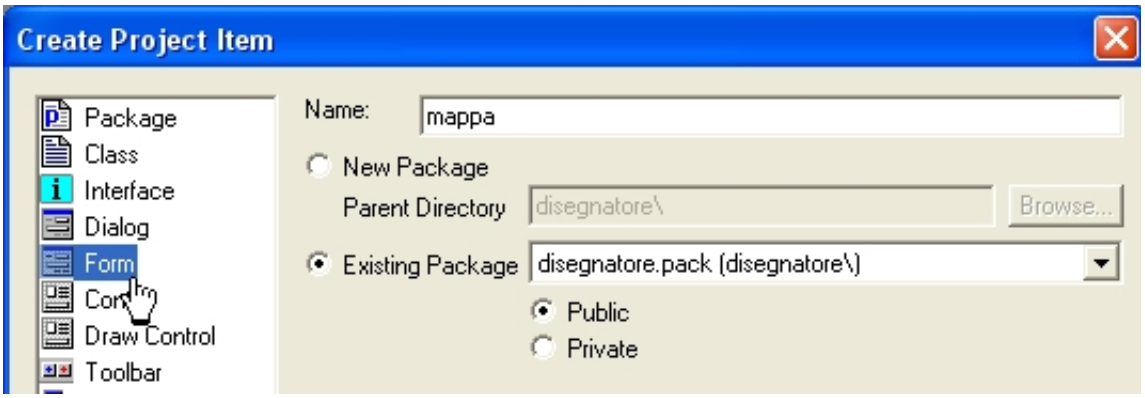

FIGURA 3.3.5. Inserire la form mappa 2

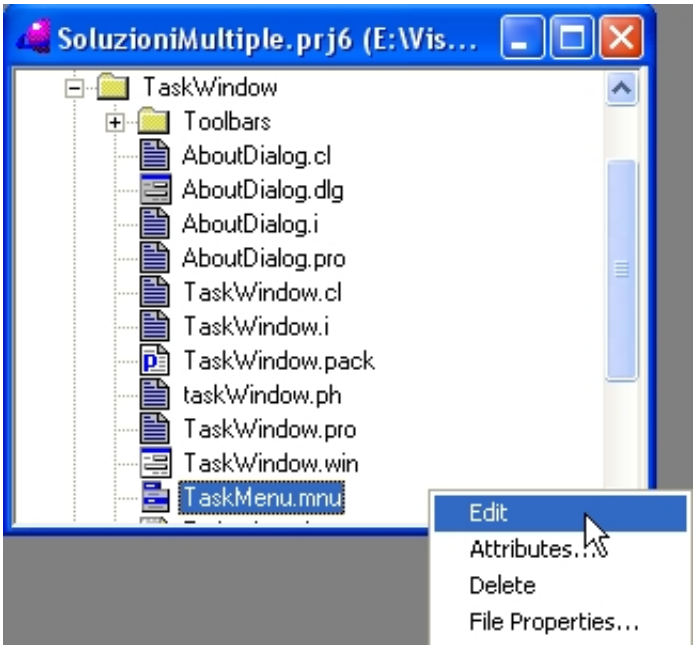

FIGURA 3.3.6. Abilitare File-New 1

L'azione precedente fa cambiare la finestra di dialogo che diventa quella della figura 3.3.10 nella pagina 84, dopo aver selezionato la voce indicata nella figura prima citata con il tasto destro fate apparire il pop-up nel quale selezionate Go to Event Code.

Fatto quanto detto prima vi apparirà una finestra con dentro il file *TaskWindow.pro*, sostituite la clausola *onFileNew(_Surce,_MenuTag):-.....* con il frammento di codice evidenziato nella figura 3.3.11 nella pagina 85. Compilate ed eseguite.

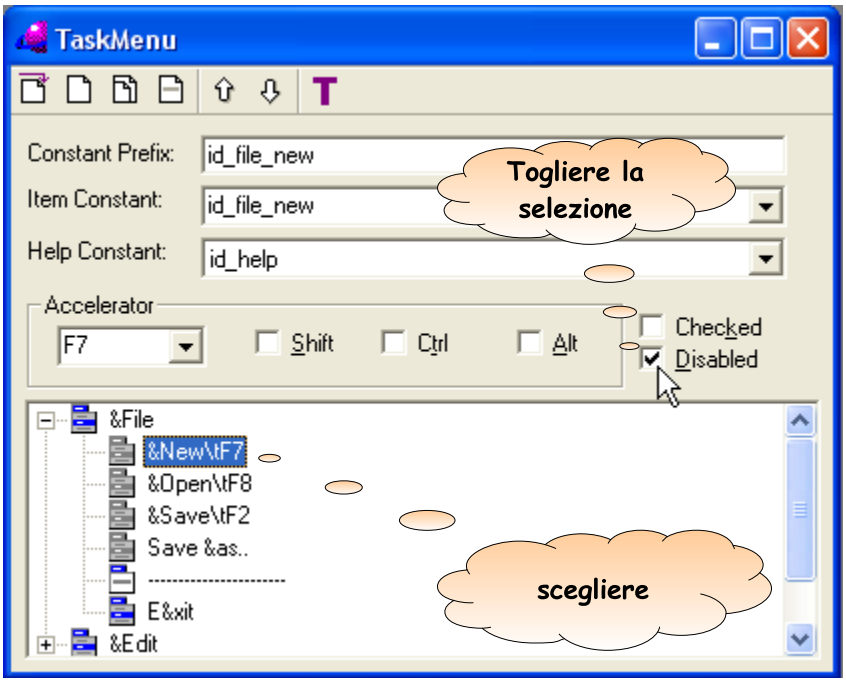

FIGURA 3.3.7. Abilitare File-New 2

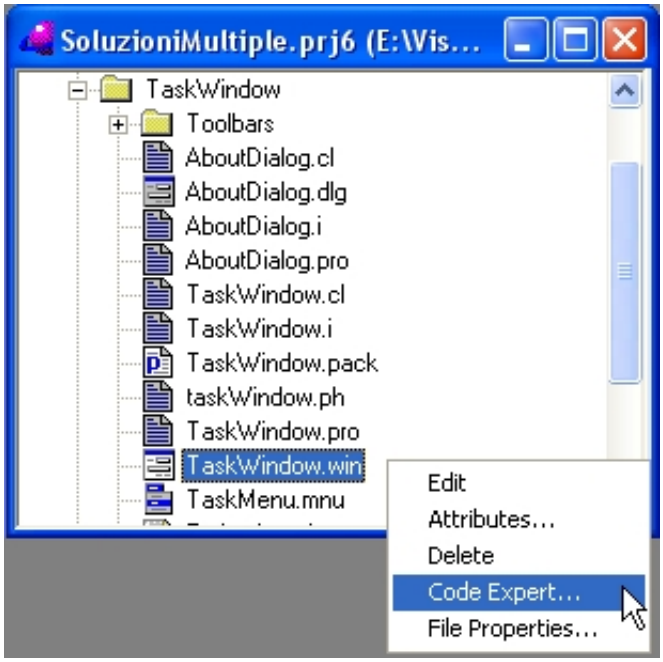

FIGURA 3.3.8. Fare apparire la form mappa 1

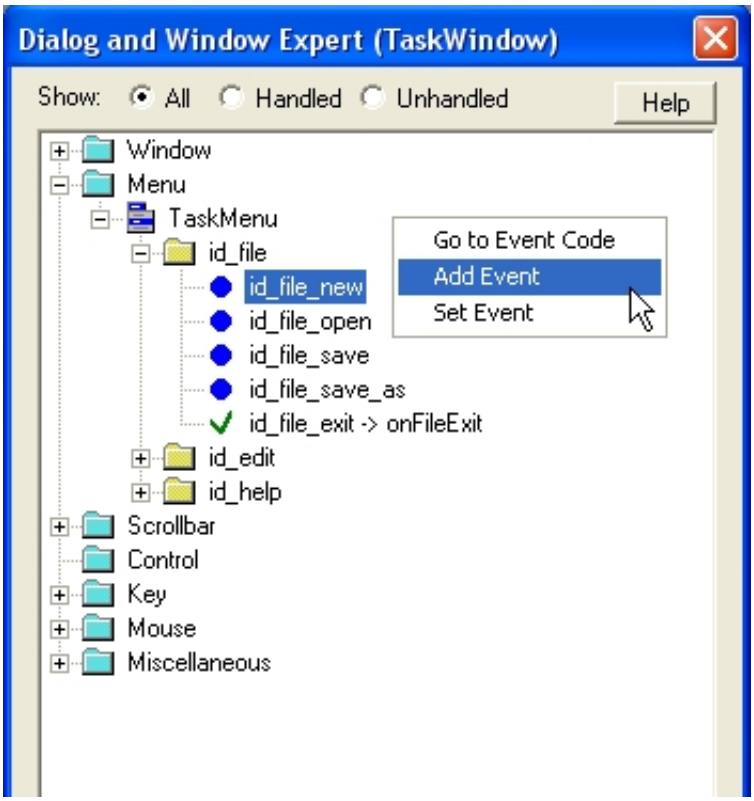

FIGURA 3.3.9. Fare apparire la form mappa 2

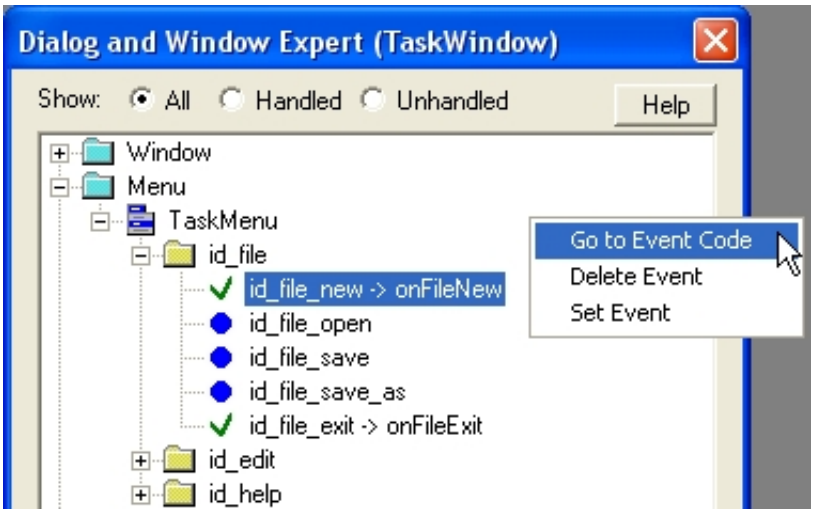

FIGURA 3.3.10. Fare apparire la form mappa 3

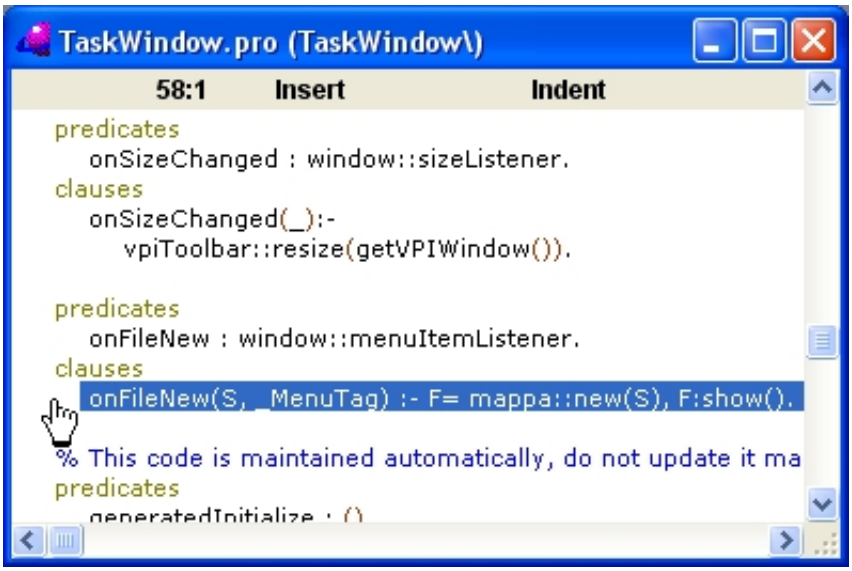

FIGURA 3.3.11. Fare apparire la form mappa 4

Costruiamo una classe simile alla classe traccia dell'esercizio precedente mettendola nel package disegnatore, di seguito nelle figure 3.3.12, 3.3.13 a fronte, 3.3.14 nella pagina 88, 3.3.15 nella pagina 89, è riportato il procedimento che bisogna seguire.

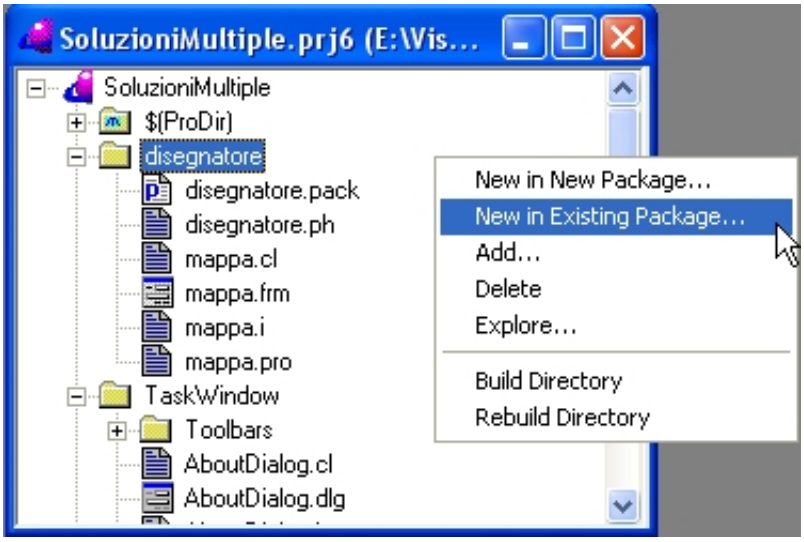

FIGURA 3.3.12. Costruire la classe *traccia* 1

Infine facciamo in modo che quando la form mappa viene disegnata venga anche disegnata la mappa delle città, per fare questo seguite le istruzioni riportate nella figura 3.3.16 nella pagina 90

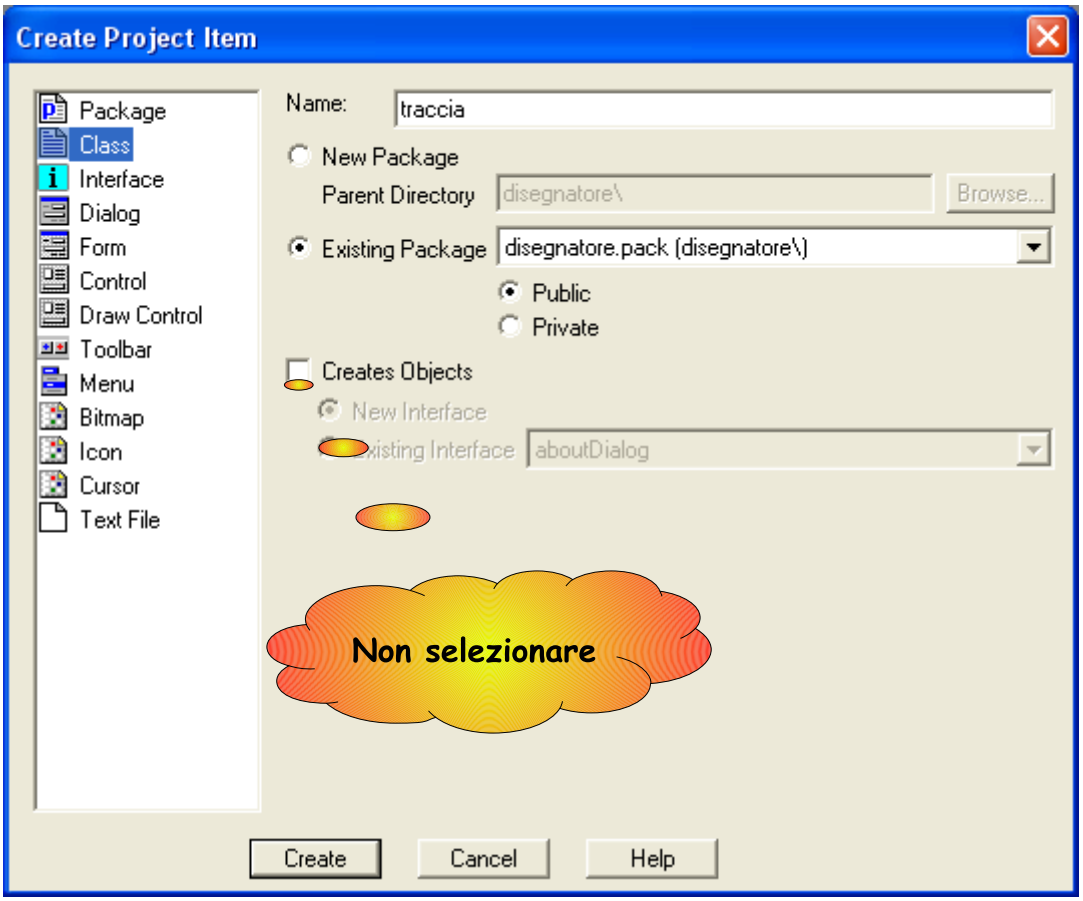

FIGURA 3.3.13. Costruire la classe *traccia* 2

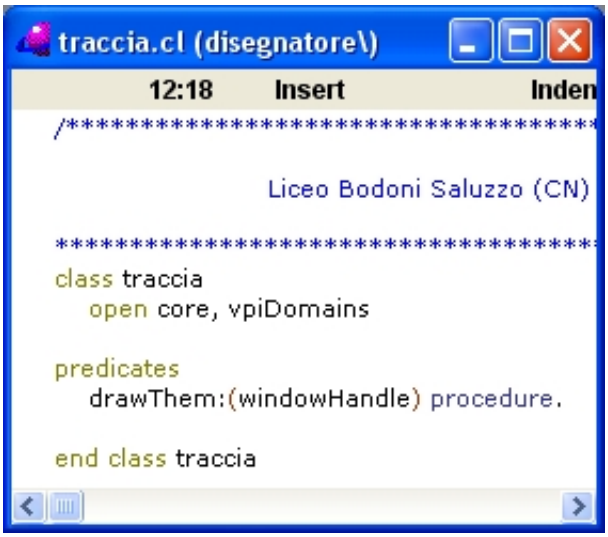

FIGURA 3.3.14. Costruire la classe *traccia* 3

traccia.pro (disegnatore\)

36:1 Insert Indent

```
/*************************************************************

                    Liceo Bodoni Saluzzo (CN)

*************************************************************

implement traccia
   open core, vpiDomains, vpi

class facts
   city:(string Name, pnt Position).
   conn:(pnt, pnt).
class predicates
   connections:( windowHandle).
   drawCities:(windowHandle).
clauses
   city("Roma", pnt(30, 40)).
   city("Firenze", pnt(100, 120)).
   city("Torino", pnt(100, 160)).
   city("Venezia", pnt(200, 100)).
   conn(pnt(30, 40) , pnt(100, 120)).
   conn(pnt(100, 120), pnt(100, 160)).
   conn(pnt(30, 40), pnt(200, 100)).

   drawCities(W) :- city(N, P), P= pnt(X1, Y1), X2= X1+10, Y2= Y1+10,
                    drawEllipse(W, rct(X1, Y1, X2, Y2)),
                    drawText(W, X1, Y1, N), fail.
   drawCities(_Win).

   connections(Win) :- conn(P1, P2), drawLine(Win, P1, P2), fail.
   connections(_Win).

   drawThem(Win) :- connections(Win), drawCities(Win).

end implement traccia
```

FIGURA 3.3.15. Costruire la classe traccia 4

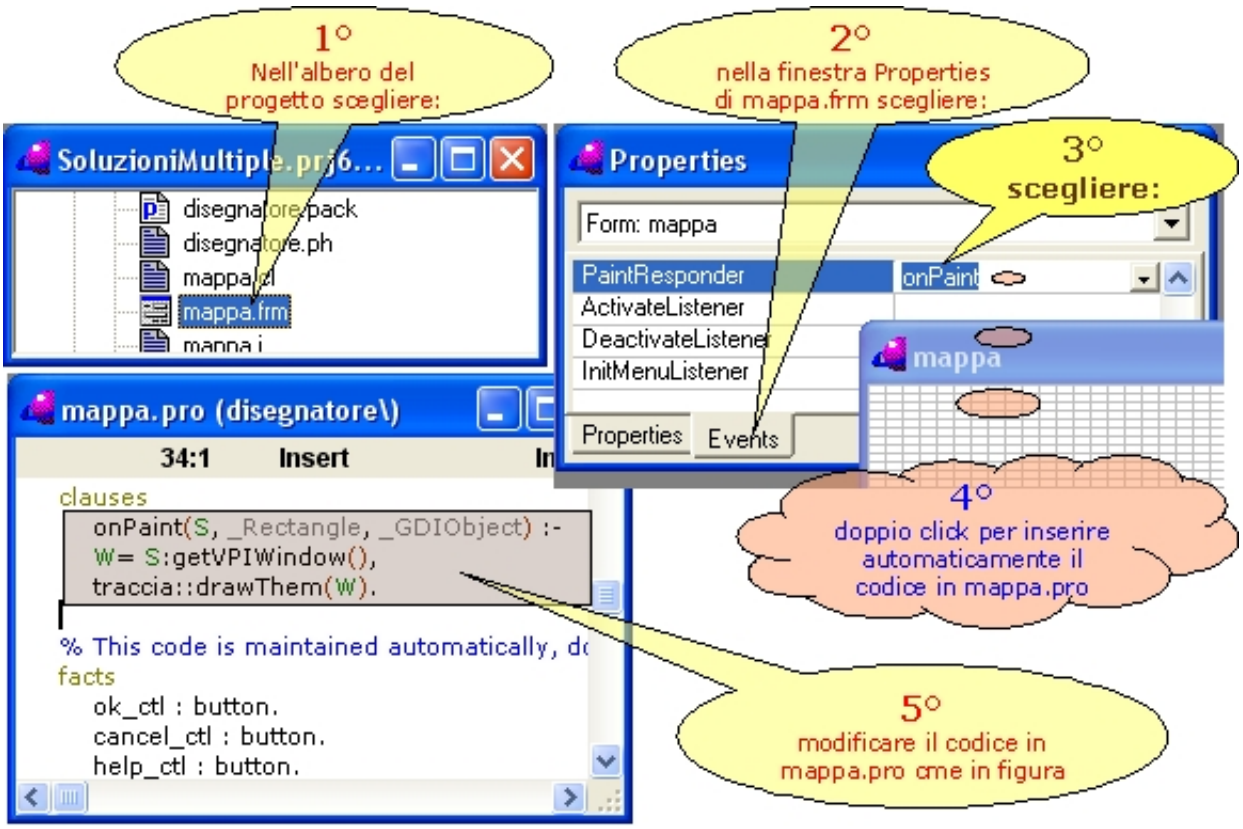

FIGURA 3.3.16. Disegnare la mappa nel form *mappa*

Per analizzare il codice scritto partiamo dal file mappa.pro, la clausola onPaint viene chiamata tutte le volte che bisogna ridisegnare la form mappa, quindi la parte interessante di questo file è quella evidenziata nella figura 3.3.16 a fronte. La procedura onPaint viene creata al momento del doppio click del punto 4 della figura 3.3.16, ciò che viene creato è uno scheletro della procedura, bisogna poi aggiungere il codice necessario per dare alla procedura le funzionalità che noi vogliamo che essa abbia. Il codice per la procedura onPaint generato automaticamente è quello riportato nel seguente riquadro

```
predicates
    onPaint : drawWindow::paintResponder.
clauses
    onPaint(_Source, _Rectangle, _GDIObject).
```

Come si può notare tutte le variabili della procedura hanno un nome preceduto dal simbolo di sottolineatura _, se una variabile ha il nome preceduto da detto simbolo e questa variabile non compare in altre parti della procedura, si chiama variabile anonima , un altro modo per indicare una variabile anonima è quello di utilizzare il semplice segno di sottolineatura. Le variabili anonime non vengono legate e possono identificarsi con qualunque cosa. Ad esempio un goal del tipo:

```
onPaint(sorgente, rettangolo,Oggetto).
```

avrebbe successo incontrando onPaint(_Source, _Rectangle, _GDIObject)., ma _*Source* non viene legata a *sorgente* e _*Rectangle* non viene legata a *rettangolo*, mentre *Oggetto* non viene legata a nulla. Ecco come nella guida viene trattato questo argomento:

> A single underscore character represents the anonymous variable. The anonymous variable is used when the value of that variable is not of interest.
> If a variable occurs only once in a clause and its name starts with an underscore, then Visual Prolog compiler will consider this variable as an anonymous variable.

La procedura onPaint è stata modificata in modo tale da ottenere prima un riferimento alla finestra che si sta disegnando e poi, usando questo riferimento viene chiamata la procedura drawThem della classe traccia, è proprio questa procedura la cosa più interessante di questo programma. La procedura drawThem è definita nel file traccia.pro, il goal drawThem(rif, finestra) può essere soddisfatto dalla regola:

```
drawThem(Win) :- connections(Win), drawCities(Win).
```

Per soddisfare il goal drawThem(Win) viene chiamato il sottogoal connection(Win) con Win legata , il sottogoal può essere soddisfatto dalla regola:

```
connections(Win) :- conn(P1, P2), drawLine(Win, P1, P2), fail.
```

La regola impone il soddisfacimento del sottogoal conn(P1,P2), nel database dei fatti il primo fatto di quel tipo che incontriamo è

```
conn(pnt(30, 40) , pnt(100, 120)).
```

Quindi P1 viene legata a pnt(30, 40) e P2 viene legata a pnt(100, 120), successivamente si assume come sottogoal drawLine(Win, P1, P2), questo è un metodo della classe vip, opportunamente importata, ed ha come effetto quello di disegnare il segmento avente per estremi P1 e P2, Successivamente viene invocato il predicato fail il quale fallisce sempre, questo fallimento provoca il backtraking il quale ci riporta al punto nel quale sono state fatte le ultime assegnazioni alle variabili, cioè a conn(P1, P2). Le due variabili vengono slegate e si cerca di soddisfare conn(P1, P2) in altro modo e precisamente con

```
conn(pnt(100, 120), pnt(100, 160)).
```

Come è facile rendersi conto, anche questa volta tutto procede come prima e anche questo goal non ha successo. Si ripete il backtraking e siccome alla fine della linea

```
connections(Win) :- conn(P1, P2), drawLine(Win, P1, P2), fail.
```

vi è la clausola fail è ovvio che questa linea non sarà in grado di soddisfare il goal connection(Win), allora si passerà alla linea successiva

```
connections(_Win).
```

questa clausola contiene la variabile anonima _Win quindi qualunque cosa sia legato a Win il goal connections(Win) ha successo, soddisfatto questo goal si ritorna a drawThem(Win) :- connections(Win), drawCities(Win). e si tenta di soddisfare il goal drawCities(Win). Come potete osservare nel file traccia.pro il codice che serve per soddisfare questo goal è molto simile, nella sua struttura, a quello usato per soddisfare connections(Win) e quindi si procederà in modo simile.

La cosa importante nelle righe del codice del file traccia,pro è l'uso della clausola fail che ci consente di iterare su un insieme di fatti facendo determinate azioni senza usare vettori di variabili e iteratori.

3.4. Le applicazioni Console

Le applicazioni Console non hanno una interfaccia grafica, non posseggono finestre e strumenti per manipolarle, girano in una finestra stile MSDos, per lanciarle dall'IDE di Visual Prolog si usa la voce di menu Build $\Longrightarrow$ Run in window .

Costruiamo un nuovo progetto usando le scelte fatte nella figura 3.4.1

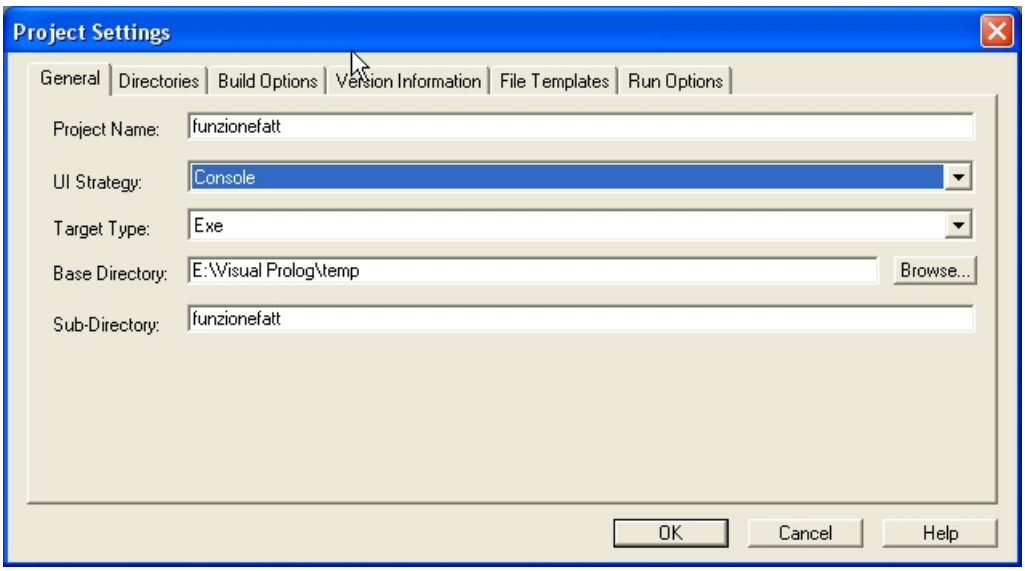

FIGURA 3.4.1. Progetto di una applicazione Console

Successivamente compilate il progetto, andate nell'albero del progetto e aprite il file main.pro che modificherete come riportato in figura 3.4.2 nella pagina seguente

Questo semplice programma contiene alcune cose interessanti: l'uso del Cut ! e la direttiva procedure (i,0) nella quarta riga del codice della figura 3.4.2 nella pagina successiva. Per spiegare il funzionamento di Cut usiamo i folletti, come abbiamo già fatto nella sezione 2.2.1 nella pagina 28, però questa volta usiamo più folletti, uno per ogni sottogoal. Consideriamo il programma appena scritto, Si parte con il goal indicato, appunto, nella sezione goal mainExe::run(main::run) , qui spunta il primo folletto che sarà il padre di tutti gli altri folletti che verranno generati nel programma. Nella casella (quella gialla) che segue è riportato il listato del programma in modo testuale per permettere il taglia-copia-incolla.

```
/*****************************************************

     Liceo Bodoni Saluzzo (CN)

*****************************************************/

implement main
    open core

class predicates
    fact:(real N, real Res) procedure  (i,o) .

clauses
    classinfo("facfun", "1.0").
    fact(N, 1) :- N<1,!.
    fact(N, N*F) :- fact(N-1, F).



    run():- console::init(),
    fact(stdio::read(), F), stdio::write(F), stdio::nl.

end implement main

goal
    mainExe::run(main::run).

```

formListe

FIGURA 3.4.2. Main.pro

```
/*********************************
      Liceo Bodoni Saluzzo (CN)
*********************************/
implement main
   open core
class predicates
   fact:(real N, real Res) procedure (i,o) .
clauses
   classinfo("facfun", "1.0").
   fact(N, 1) :- N<1,!.
   fact(N, N*F) :- fact(N-1, F).
   run():- console::init(),
   fact(stdio::read(), F), stdio::write(F), stdio::nl.
end implement main
goal
   mainExe::run(main::run).
```

Riprendiamo il discorso dei folletti, il goal, cioè l'antenato di tutti i folletti genera,

grazie all'istruzione della riga 26 figura 3.4.2 nella pagina precedente, genera un folletto figlio il folletto run(), e rimane ancorato all'inizio delle riga 26 in attesa che il suo folletto figlio abbia successo. Il nuovo folletto comincia a svolazzare sul codice partendo dall'alto e muovendosi verso il basso, percorrendo ogni riga da sinistra verso destra, cercando una clausola la cui testa sia run(). La clausola cercata viene trovata alla riga 20 figura 3.4.2 a fronte, il folletto run() si ancora all'inizio della riga 20 e comincia a produrre figli. Dapprima produce il folletto console::init() , questo frammento di codice è stato prodotto automaticamente da CodeExpert e riguarda una delle procedure della PrologFoundationClass, questa procedura ha sempre successo e quindi il folletto console::init() comunica a run() che la cosa è andata a buon fine . Il folletto run() annota il successo di console::init() e annota anche che non vi è una seconda possibilità di soddisfare console::init() perché questa è una procedura e quindi ha sempre successo e in un solo modo. A questo punto il folletto run() genera un secondo figlio il folletto fact(stdio::read(),F) . Il folletto fact(stdio::read(),F) è una procedura definita alla riga 11 della figura 3.4.2 nella pagina precedente, quella dichiarazione ci dice che abbiamo a che fare con una procedura i cui due parametri sono del tipo (i,o) cioè il primo parametro deve essere legato, quindi non è ammessa una variabile in quella posizione, il secondo invece deve essere una variabile . Inoltre questo folletto è una procedura, deve quindi sempre avere successo e non ci deve essere possibilità di backtracking quindi vi deve essere una sola possibilità di successo, questo significa che ogni clausola la cui testa si identifica con fact(_i, _o) o termina con il cut ! oppure è l'ultima clausola di quel tipo. Il nostro folletto parte quando introduciamo un numero da tastiera e quindi riempiamo il suo primo parametro con quel numero mentre il secondo resta libero.Il folletto comincia a svolazzare sul programma e, partendo dall'alto, incontra la riga 15 della figura 3.4.2 a fronte: fact(N, 1) :- N<1,!. lega la variabile N al numero che abbiamo introdotto da tastiera e la variabile F a 1, successivamente si ancora alla testa della riga 15 ed emette un folletto figlio N<1. Questi avrà successo se abbiamo introdotto un numero minore di uno, in tal caso il folletto N<1 comunica il successo al folletto fact(stdio::read(),F) poi si autodistrugge, il folletto fact(stdio::read(),F) emette allora il nuovo folletto ! il quale ha sempre successo ma ha un effetto deleterio su suo padre: gli inibisce il backtracking, a questo punto fact(stdio::read(),F) comunica a run() che ha avuto successo con F=1 e che non sono possibili altre soluzioni, quindi run() emette altri folletti che stampano F sullo schermo e mandano il cursore a capo.

Se da tastiera abbiamo introdotto un numero maggiore o uguale a uno il folletto N<1 non avrà successo quindi il folletto fact(stdio::read(),F) non potendo essere soddisfatto dalla riga 15 passerà alla 16 la quale contiene una clausola ricorsiva, non ci interessano in questo momento le clausole ricorsive l'unica osservazione da fare è che il primo parametro ad ogni passo viene diminuito di una unità, alla fine si ritornerà necessariamente alla riga 15 la quale inibirà il backtraking di fact assicurandoci che questa clausola è proprio una procedura.

3.5. Menù - toolbar

Vogliamo scrivere un programma in grado di manipolare le liste [**1**]. Impostiamo un progetto utilizzando la solita procedura, come illustrato nella figura 3.5.1

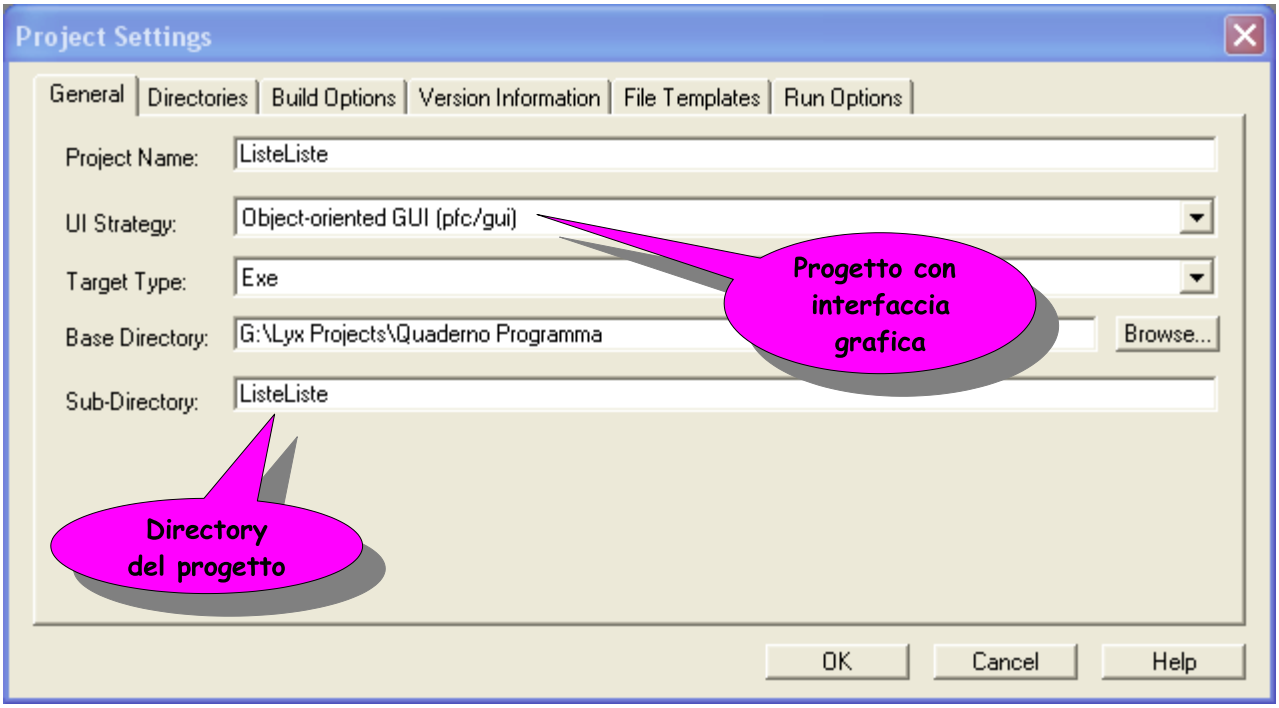

FIGURA 3.5.1. Progetto ListeListe

Costruiamo un nuovo Package come descritto in figura 3.5.2 a fronte

Nel package appena costruito ci mettiamo una nuova form come descritto nella figura 3.5.3 nella pagina 98

Nella form appena creata aggiungiamo una ListEdit e un pulsante, come indicato nella figura 3.5.4 nella pagina 99

Per lanciare la form usiamo il menu e la barra delle icone, il menu lo si può modificare usando l'opzione indicata nella figura 3.5.5 nella pagina 99.

Il doppio click sull'opzione scelta produce la finestra di dialogo riportata nella figura 3.5.6 nella pagina 100, utilizzando questa finestra come è indicato si aggiunge un nuovo item al menu, dentro l'item bisogna mettere i sotto item , per fare ciò bisogna selezionare l'item appena inserito e procedere come indicato nella figura 3.5.7 nella pagina 100

Come è noto le opzioni del menu sono ripetute nella toolbar, per rendere il tutto più semplice abbiamo tolto alcune icone inutili dalla toolbar, per fare questo bisogna innanzitutto richiamare la toolbar, nella finestra del progetto selezionare ProjectToolbar.tb come è indicato nella figura 3.5.8 nella pagina 101 , comparirà quanto riportato nella

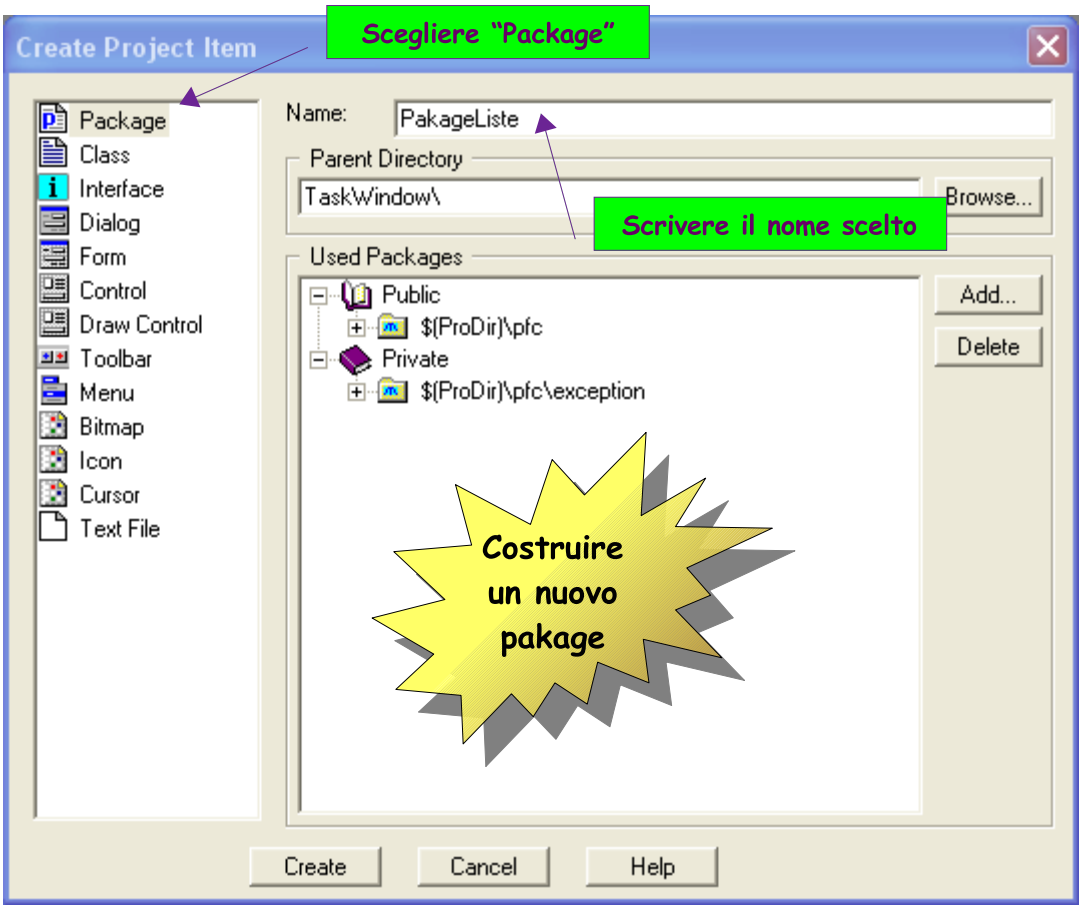

FIGURA 3.5.2. Package PackageListe

figura 3.5.9 nella pagina 101. I controlli possono essere aggiunti alla toolbar posizionando,con gli appositi tasti, il cursore verde nel punto voluto e poi selezionare il controllo scelto.

Dobbiamo ancora spiegare come abbiamo ottenuto la nuova icona che appare nella toolbar, per creare una nuova icona scegliere New ad esempio dal menu File, avendo avuto cura di scegliere nella finestra dell'albero del progetto Toolbar, nella finestra che appare scegliere Bitmap, dare un nome alla nuova bitmap ed infine scegliere Create, vedi figura 3.5.10 nella pagina 102. Comparirà la finestra degli attributi della nuova icona, correggere l'altezza e la larghezza come indicato in figura 3.5.11 nella pagina 102. Scegliendo Ok comparirà la finestra riportata in figura 3.5.12 nella pagina 103.

Dopo aver disegnato la bitmap salvarla usando il tasto Salva come è indicato in figura 3.5.12 nella pagina 103

Per aggiungere la nuova bitmap alla Toolbar procedere come indicato nella figura 3.5.13 nella pagina 103, di solito una icona della toolbar ha lo stesso comportamento di una

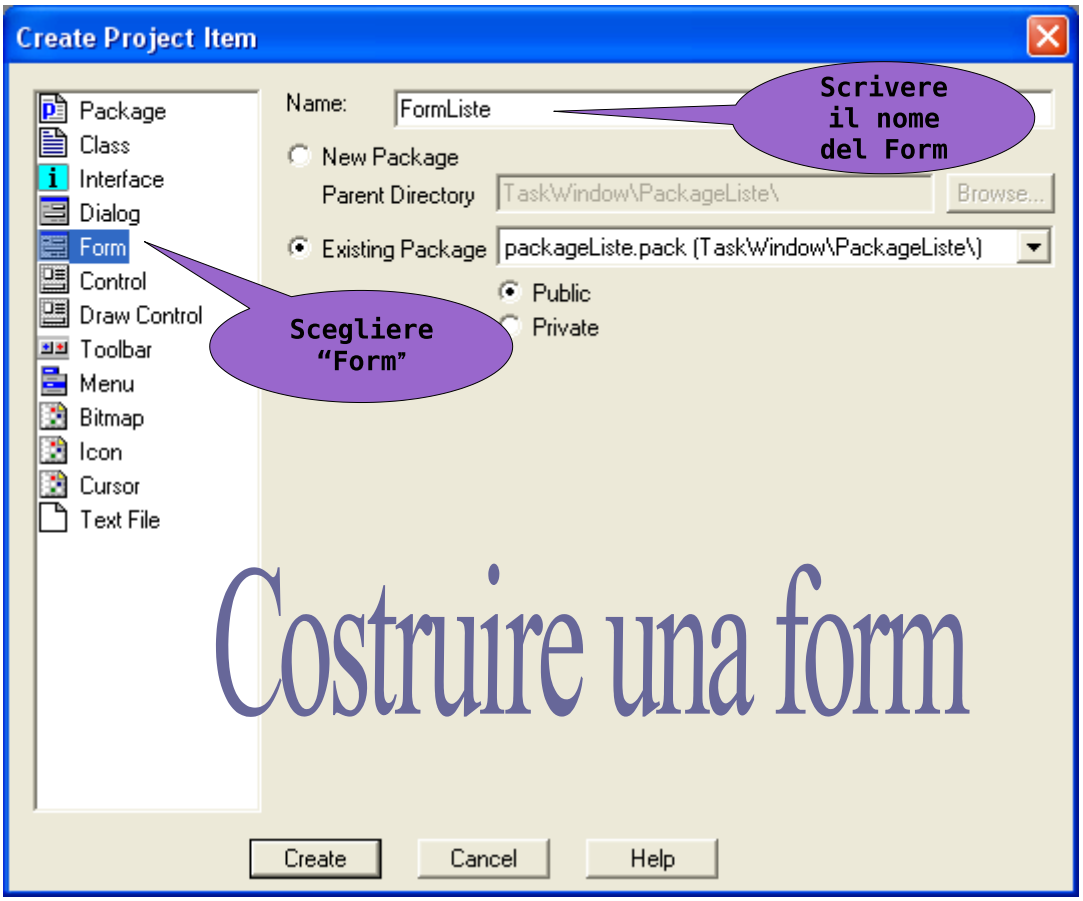

FIGURA 3.5.3. Form FormListe

corrispondente voce del menu, affinché questo avvenga è necessario che abbiano lo stesso identificatore (vedi figura 3.5.13 nella pagina 103 punto 3)

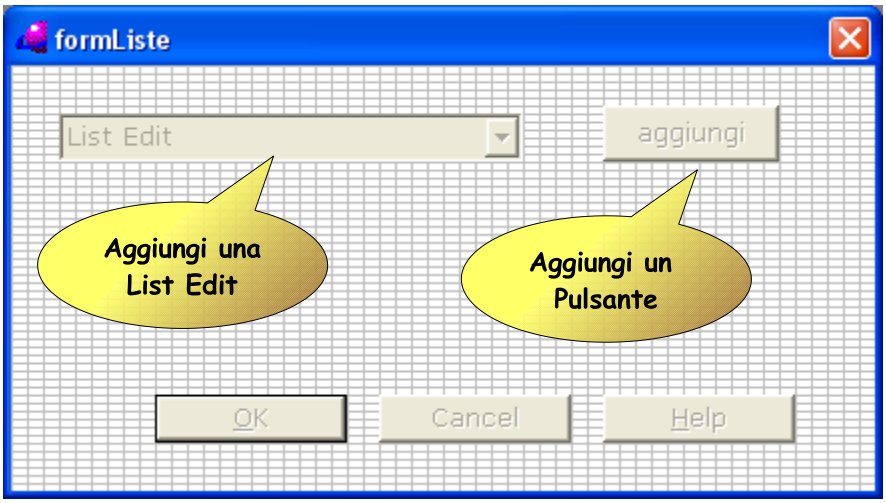

FIGURA 3.5.4. La form completa

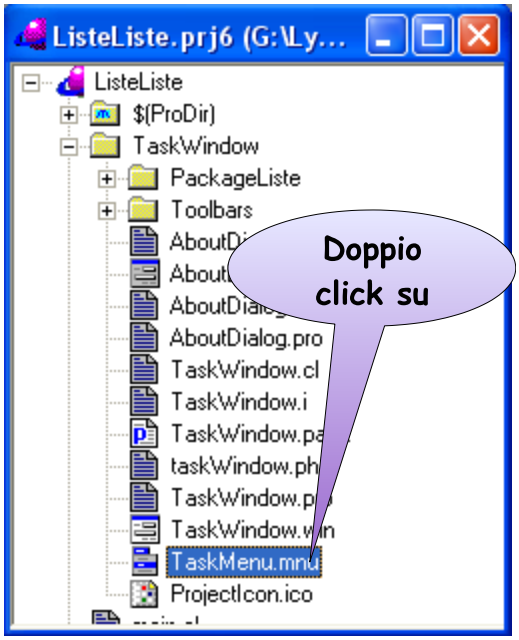

FIGURA 3.5.5. Menu

FIGURA 3.5.6. Aggiungere un item al menu

FIGURA 3.5.7. Aggiungere un sotto item

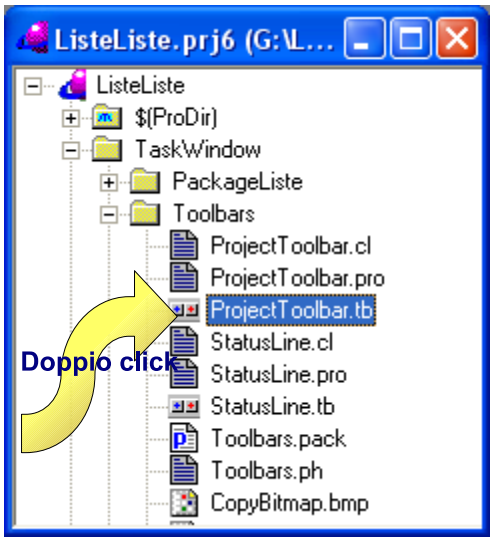

FIGURA 3.5.8. Richiamare la toolbar

FIGURA 3.5.9. Toolbar e i suoi controlli

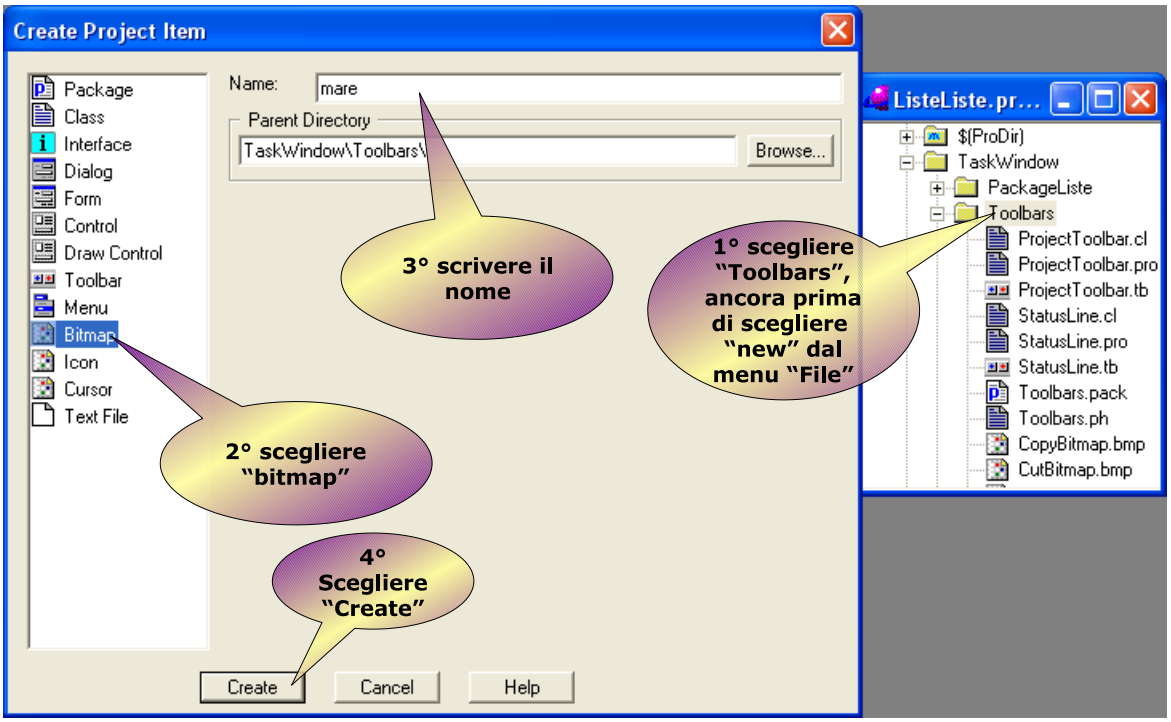

FIGURA 3.5.10. Creare una nuova icona, primo passo

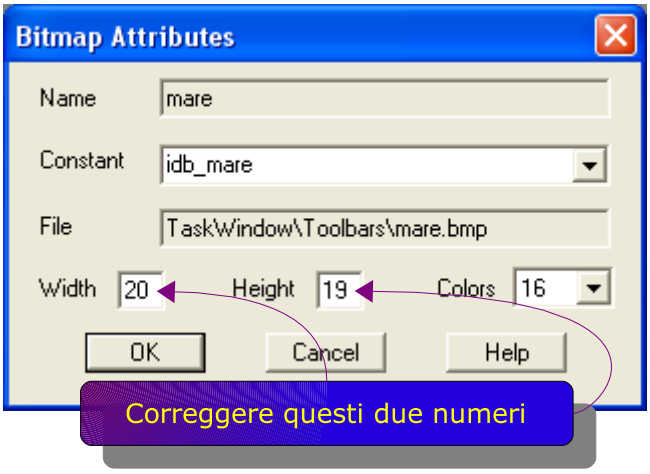

FIGURA 3.5.11. Attributi della nuova icona

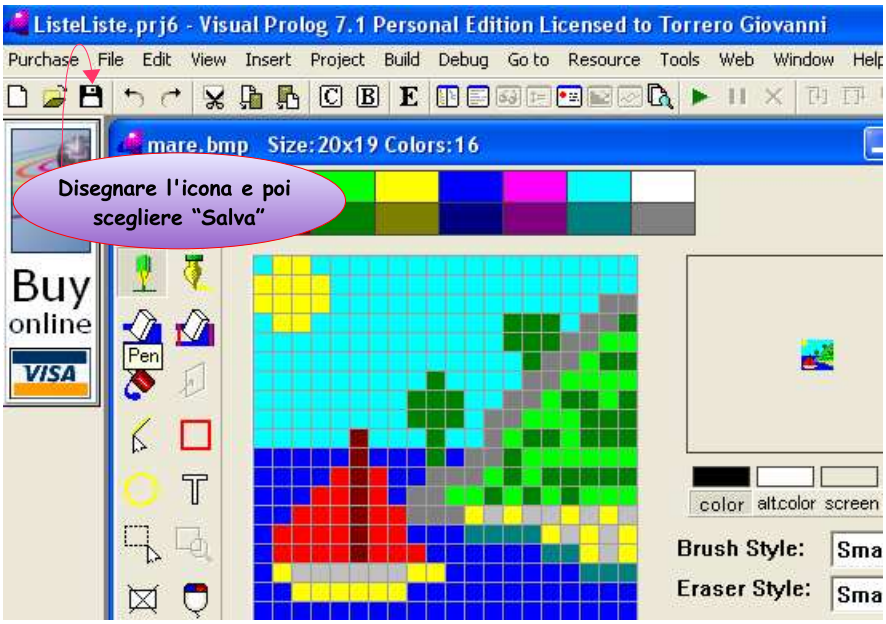

FIGURA 3.5.12. Disegnare la nuova icona

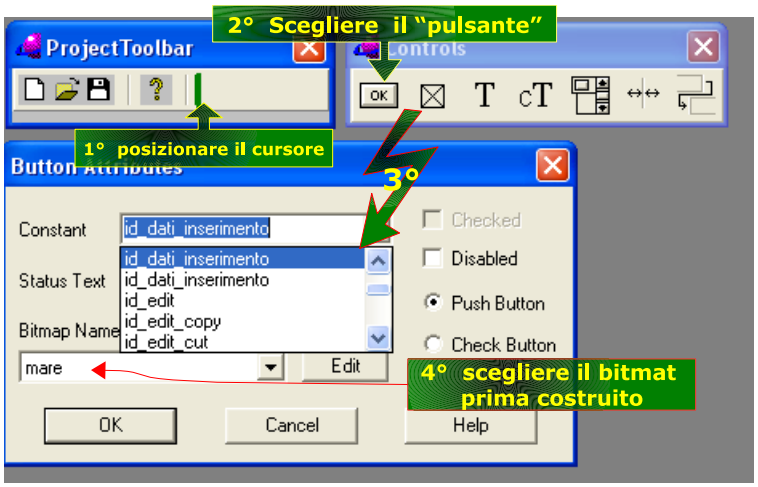

FIGURA 3.5.13. Aggiungere un pulsante alla Toolbar

FIGURA 3.5.14. Aspetto finale della Toolbar

Dobbiamo ora collegare la voce del menu e la toolbar ad un particolare comportamento, in particolare vogliamo che quando si seleziona Dati - Inserimento, oppure l'icona inserita nella toolbar, compaia la form formListe . Per fare questo richiamate l'albero del progetto e procedete come indicato nella figura 3.5.15 , si aprirà una finestra riportata nella figura 3.5.16 nella pagina successiva, successivamente sceglieremo Go To Event Code per raggiungere la parte di codice creata automaticamente, questo frammento di codice va modificato come indicato in figura 3.5.17 a fronte. Se compiliamo ed eseguiamo il programma possiamo controllare il suo funzionamento.

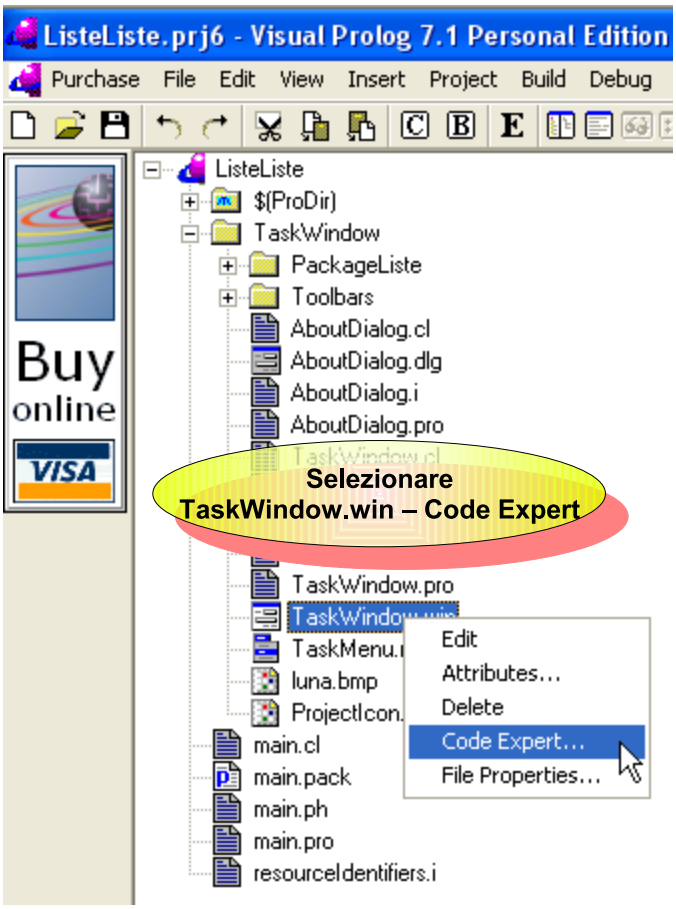

FIGURA 3.5.15. Richiamare Code Expert

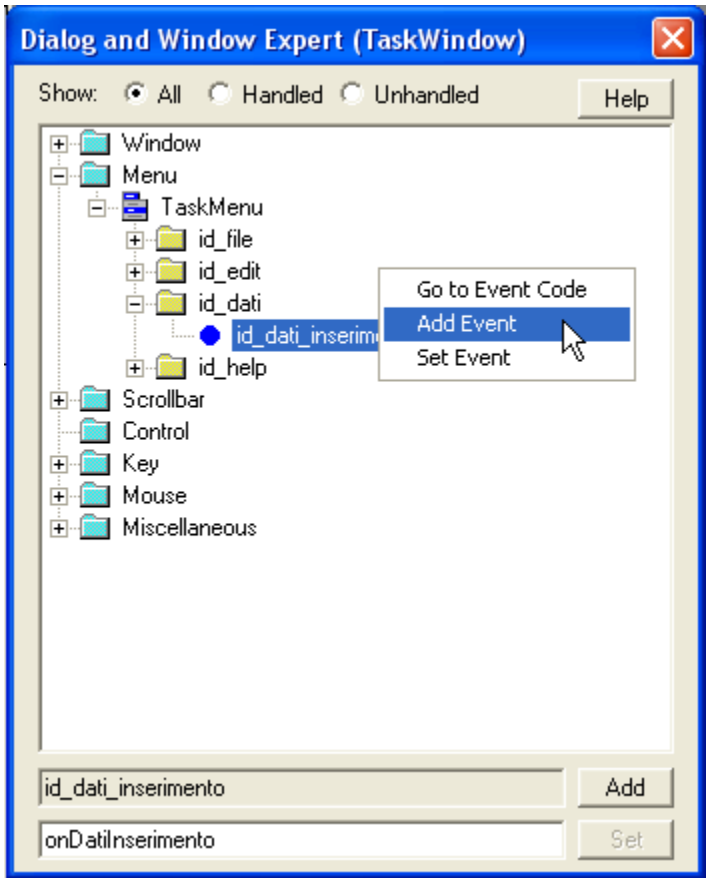

FIGURA 3.5.16. Code Expert in azione

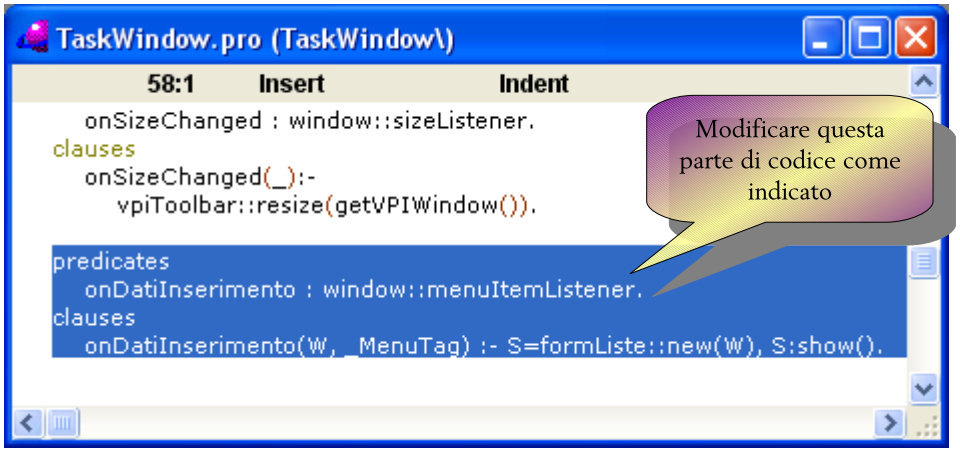

FIGURA 3.5.17. Modifiche al codice generato da Code Expert

Ora ci dobbiamo occupare della formListe, in particolare vogliamo conoscere qualche cosa sulla List Edit. Per ottenere queste informazioni andiamo sull'Help, nel Sommario scegliamo Prolog Foudation Classes - gui - controls - List Edit List Edit compare due volte sia nelle interfacce che nelle classi, la questione delle classi e delle interfacce merita un approfondimento maggiore, per ora non vogliamo entrare nel dettaglio ma nell'interfaccia troviamo i metodi della classe che è possibile usare, un metodo interessante della classe List Edit è il seguente:

listControl::add/1 add : (string String).
Appends a new string item String to the end of the list in the list control.

Useremo questo metodo per aggiungere elementi, e quindi popolare, la lista di List Edit. Useremo anche altri metodi per dare maggiore efficienza al controllo, di seguito è riportato quanto detto su questi metodi dalla guida:

window::setText/1 setText : (string Title).
Sets the specified title (caption) text to the current window.

window::getText/0-> getText : () -> string Title.
Returns the title (caption) Title of the current window.

window::setFocus/0 setFocus : () procedure ().
Sets an input focus to the current window.

Diamo funzionalità al pulsante che abbiamo inserito, tramite l'albero de progetto richiamiamo formListe.frm come è indicato nella figura 3.5.18

FIGURA 3.5.18. Richiamare formListe.frm

Selezioniamo il pulsante aggiungi, nella finestra Properties selezioniamo la scheda Events, scegliamo l'evento ClickResponder, sulla destra compare il nome del corrispondente metodo OnPushButtomClick (vedi figura 3.5.19 a fronte), facendo doppio click su questo nome veniamo proiettati in formListe.pro.

Code expert produce uno scheletro del metodo OnPushButtomClick, noi dobbiamo modificarlo come indicato in figura 3.5.20 nella pagina successiva

FIGURA 3.5.19. Lanciamo Code Expert

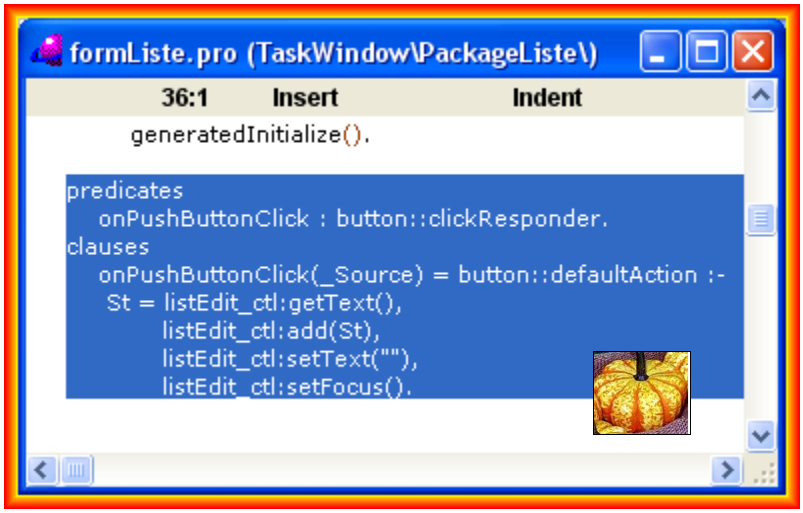

FIGURA 3.5.20. Modificare OnPushButtomClick

Indice analitico

backtracking, 25
base di conoscenze, 19
brush, 74

classe, 39, 50
classi, 37
clausola, 23
clausole, 18
core, 74
Create Project Item, 39

fatti, 19

oggetti, 37
open, 74

package, 37, 38
patStyle, 74
predicati, 19
Project Window, 39

regola, 19

se, 19

TaskWindow, 39
teoria, 19

variabile anonima, 91
vpiDomains, 74

windowHandle, 77

Bibliografia

[tyros] Eduardo Costa "Visual Prolog for tyros"
URL http://www.visual-prolog.com/vip/tutorial/tut14/default.htm,
inserito nel sito Visual Prolog http://www.visual-prolog.com/ 2

[tutorial] Sabu Francis "Fundamental Prolog" e "Fundamental Visual Prolog"
URL Tutorial http://www.visual-prolog.com/vip/tutorial/

Bibliografia

[1] P. R. Robinson. *Programmazione in turbo prolog*. Informatica. McGraw-Hill, McGraw-Hill Libri Italia - Piazza Emilia 5 - 20129 Milano, 1987. 25, 96