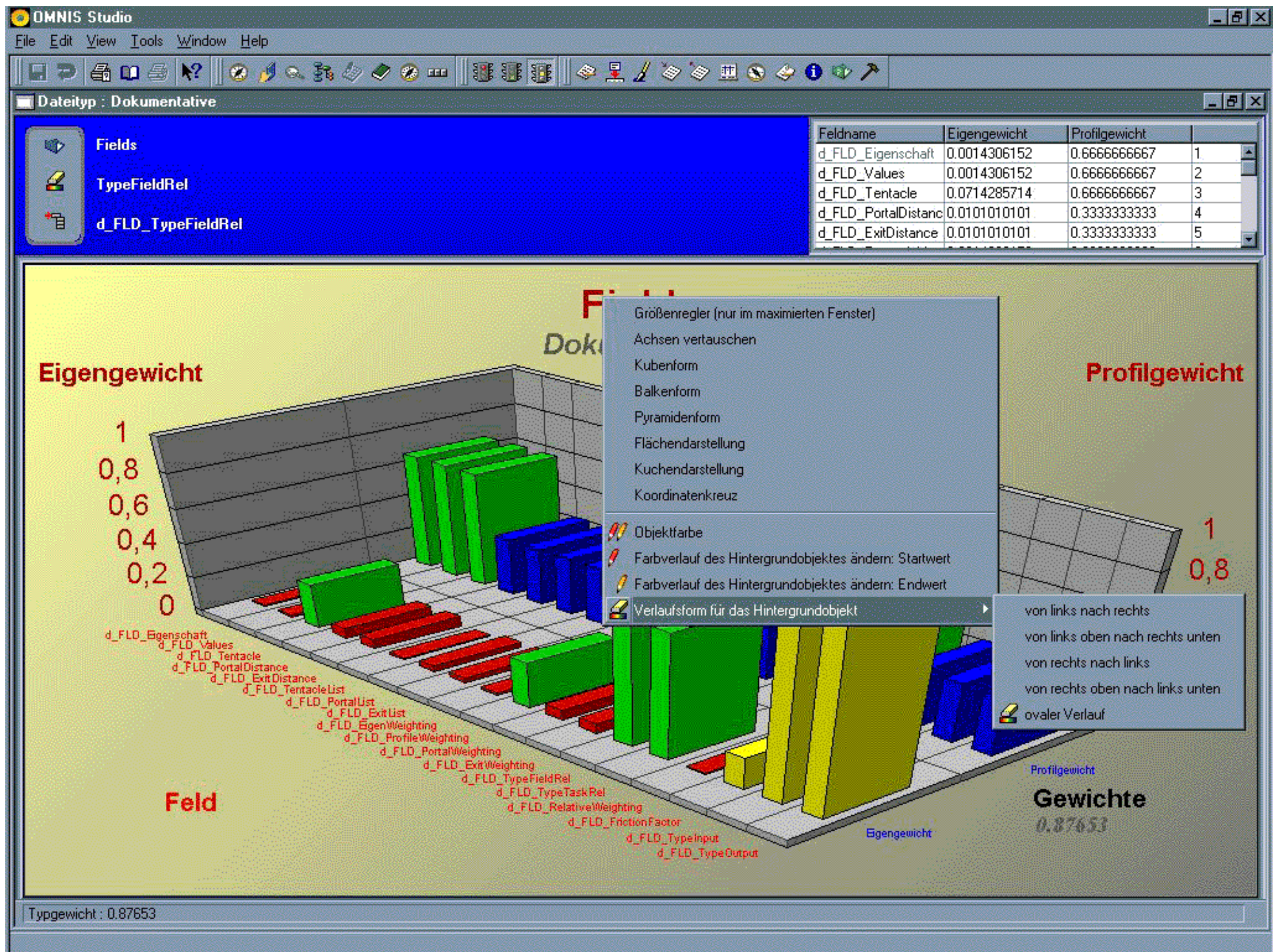

Abbildung 245 Rechte Maus in Aktion (bei maximiertem Fenster)

Im Zuge der Aktionen, die die Rechte Maus tun soll, biete ich auch dieses nette Regelpult an, mit dem sich die Graphik verbiegen und verschieben lässt. Da dieses Pult jedoch für ein kleines Fenster zu groß ist, möchte ich es nur dann tun, wenn das Fenster maximiert ist. Das wiederum bedeutet aber, dass ich sicherstellen sollte, dass dieses Pult im anderen Fall immer unsichtbar ist. Also ergänze ich den Teil der Fenster-Methode **$ChangeGraphic**, der den Normalfall regelt, wenn **pEventcode** also nicht **evMaximized** ist, zu:

w_Navigator, $ChangeGraphic

Calculate #F as $cinst.$objs.File.$showsliders.$assign(kFalse)

Weiterhin füge ich noch zwei Auswahlen in meiner kleinen **AddGraphicChanges** ein:

w_Navigator, AddGraphicChanges

Calculate l_Key as 'ShowWall'

Calculate #F as i_LST_RGraphic.$add()

Calculate l_Key as 'Stretch'

Calculate #F as i_LST_RGraphic.$add()

Mit der ersten möchte ich die Wände entfernen lassen, mit der zweiten die Möglichkeit geben, den Dehnungsfaktor, den ich beim Eigengewicht verwendet habe, auszuschalten.

In derselben Methode **AddGraphicChanges** muss ich übrigens noch den Wert „Cubes“ in „Cube“ umändern, ebenfalls den Text dazu in der **$GiveText** der **Startup_Task**. Dann passt er nämlich zu den Konstanten **kGR3D**, die Omnis zur Kennzeichnung der verschiedenen 3D-Varianten vorweist.

Nun bekomme ich jedoch ein kleines Problem mit meinem Farb-Button **Color**. Er verschwindet manchmal einfach deshalb, weil er seinen Platz mit einem anderen Objekt teilt, der Graphik-Komponente. Ist letztere aktiv, ist ersterer verschwunden. Ich versuche es mit

TEST

Calculate $cinst.$objs.Color.$hasfocus as kTrue

Ähnlich **$ctarget** bringt es das gewünschte Objekt in den Aufmerksamkeitsbereich von Omnis, doch richtig zuverlässig bleibt der Button nicht im Vordergrund, die Graphik-Komponente ist viel zu lebhaft. Also setze ich ihn generell auf „unsichtbar“, per **PROPERTY MANAGER**. Ich werde ihn also nur dann sichtbar werden lassen, wenn Farbänderungen gewünscht sind, wenn also die Rechte Maus mit einer der „Color“-Varianten vorstellig wird.

Nun muss ich bei den Farbgebungen noch daran denken, dass ich verschiedene Varianten von Farbgebungen habe, Objekte, Texte, die **Wash Control**-Komponenten. Um immer ein ganz bestimmtes Graphik-Objekt zu gewährleisten, füge ich eine neue Instanzvariable mit dem Namen des **$background**-Objektes der Graphik ein und lese dieses Objekt in meiner Fenster-Methode **$construct** ein.

w_Navigator,
$construct

Calculate i_DefaultGraphicObj as '$background'

Do $cinst.$SetGraphicObject(i_DefaultGraphicObj)

Damit kann auch die Feldmethode des Graphik-Objektes **File** reduziert werden. Dort nämlich tue ich nun ebenfalls nichts weiter mehr, als diese neue Methode einzulesen. Damit sieht **$event** von **File** insgesamt kurz und bündig aus:

w_Navigator,
File

On evGraphObjChange

Do $cinst.$SetGraphicObject()

On evRMouseDown

Do $cinst.$DoRightMouse(,kTrue)

Die neue Methode ermittelt dann das gewünschte Graphik-Objekt der Komponente und setzt den Farb-Wert des **Push Button Color** auf den jeweiligen Vorgabewert. Beim **$background**-Objekt soll es der helle Wert der **Wash Control**-Komponente sein, bei ausgesuchten Objekten die Schriftfarbe, ansonsten jedoch die Flächenfarbe, wenn es nicht gerade die Objekte der Werteanzeige sind. Diese werden nach einem wählbaren Modell eingefärbt, also übergehe ich die Anzeige der aktuellen Farbe einfach. Dabei ist an dieser Stelle noch völlig uninteressant, ob der Button überhaupt sichtbar ist. Woher ich die Objekte-Bezeichnungen habe? Aus dem **EXAMPLE** - dort werden die Namen mit jedem Klick hübsch angezeigt.

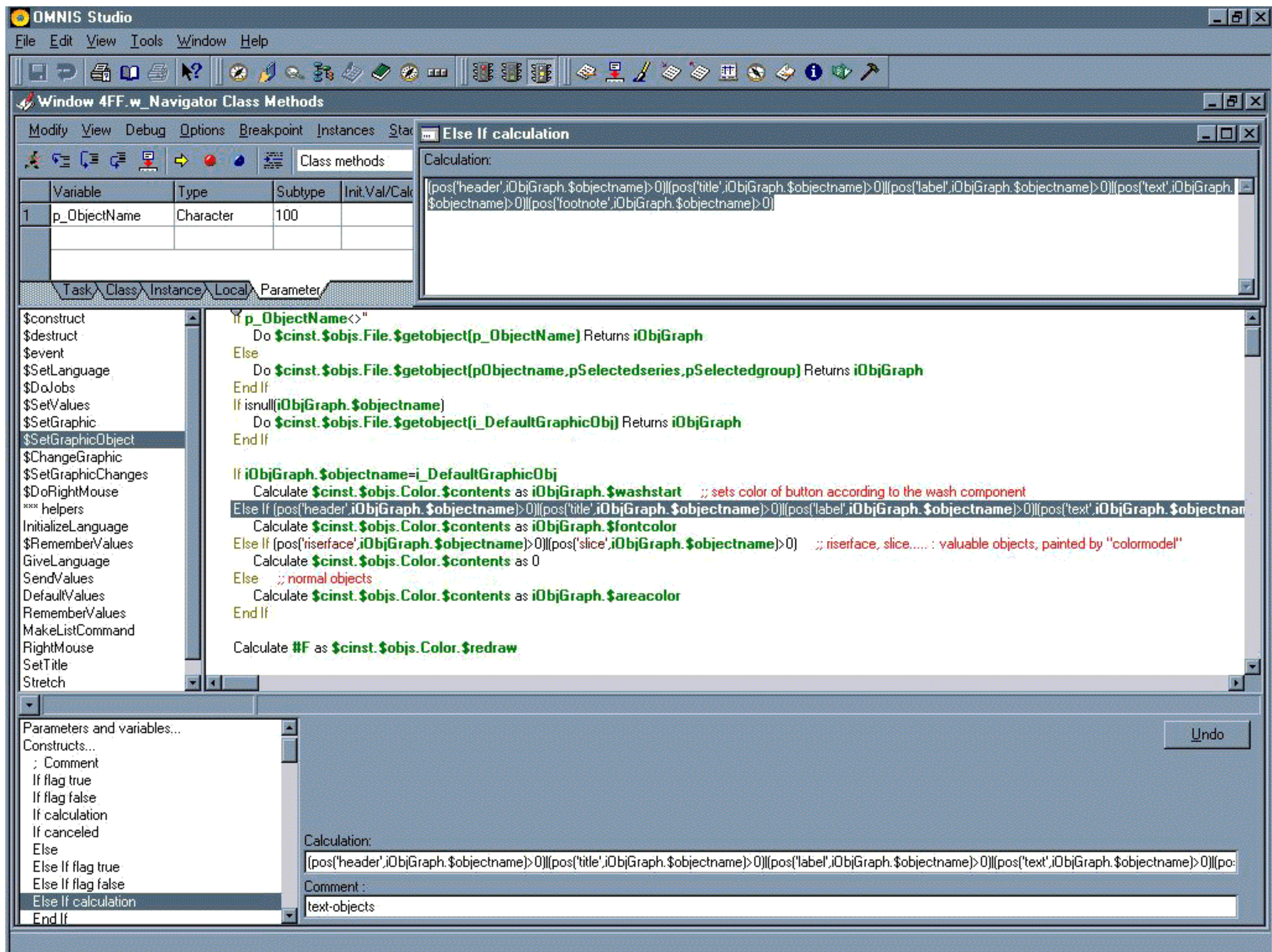

Abbildung 246 Fenster-Methode $SetGraphicObject

Damit habe ich das Problem der Sichtbarkeit des Buttons jedoch nicht wirklich gelöst, schließlich wollen meine Anwender ja nicht gezwungen werden, hastig auf Button-Jagd zu gehen, sondern mal dieses und mal jenes dazwischen erledigen. Dieses oder jenes lässt den Button aber gelegentlich verschwinden, die Graphik-Komponente hat ganz selbstverständlich Vorrang, also verschiebe ich ihn in den Kopfbereich. Wohin bloß in dem kleinen Fenster? Da ich keinen passenden Ort finden kann, lege ich ihn über die drei ersten Buttons und zwar so, dass er diese völlig bedeckt. Er ist ja meist unsichtbar, doch wenn er dann sichtbar ist, möchte ich nicht, dass die anderen noch an allen Ecken und Enden hervorlugen.

Diese Umordnung ist jetzt fast nicht mehr hinzukriegen im „normalen“ Fenster-Layout. Ich maximiere deshalb im Design-Modus mein Fenster, dann kann ich den Button schön über die anderen drei **Push Button** legen und so lange ziehen und wenden, bis er groß genug und richtig positioniert ist. Bloß nicht vergessen, im Design-Modus dann auch wieder auf die Normalgröße zurückzukehren! Kommt ein Speichervorgang dazwischen, nimmt Omnis diese Größe als „Normal“ an und das ist gar nicht im Sinne der Erfinder.

Damit der **Push Button Color** sich bei einer Fenstergrößen-Änderung nicht vondannen macht, erhält er auch die Einstellung „edgefloat“=**kEFnone**. Um ihn dann noch deutlich von den anderen Buttons abzuheben, gebe ich ihm wieder ein richtiges Button-Aussehen: „buttonstyle“ = **k3DSystemButton**, sowie eine passende blaue Oberfläche, da Icons bei dieser Sonderausstattung „colorpicker“ nicht akzeptiert werden.

Diese Lösung hat jedoch noch einen Haken - nun muss ich den Button wieder abschalten können. Das werde ich jedoch einfach durch Aktivierung des Buttons tun. Wenn ihn die Anwender einmal benutzt haben, soll er wieder verschwinden, doch das ist nicht Sache meiner Rechte-Maus-Verarbeitung, wird deshalb im Moment vertagt.

Bevor ich an die Aufgaben des zweiten Menüs komme, die Änderung der Verlaufsformen, ergänze ich noch die Möglichkeit, den Dehnungsfaktor für das Eigengewicht aus- und einzuschalten sowie die Wände anzuzeigen oder zu verbergen. Letzteres wird über das Attribut **$showwall** gesteuert, das drei Ausprägungen hat: **kGRfloor**, **kGRleft**, **kGRright** aufweist. Deren Kombinationen sind als einfache Zahlen dargestellt, deren Wirkung mit dem Stepper wunderschön beobachtet werden kann. Ich möchte aber nur zwei Kombinationen haben: gar keine Wände = 0 und alle Wände = 7.

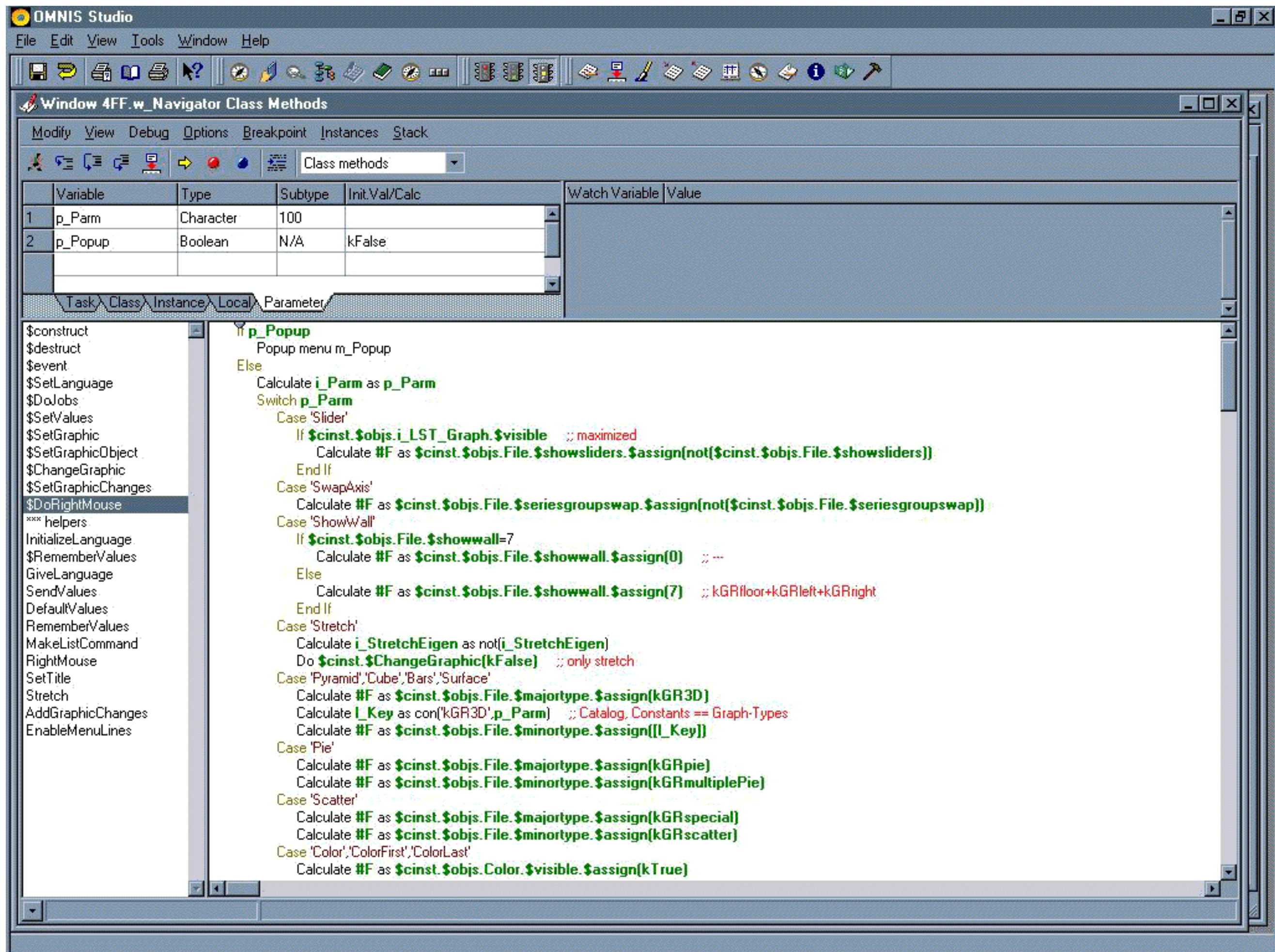

Abbildung 247 Erster Teil der $DoRightMouse

Die Routine **$ChangeGraphic** muss in diesem Zusammenhang um den bei „Stretch" erkennbaren boole'schen Parameter **p_NotOnlyStretch** ergänzt werden, der unterlassungsmäßig auf **kTrue** gestellt wird. Dieser Parameter soll nun Voraussetzung dafür sein, dass die Graphik-Datenliste **iListGraph** beschriftungsmäßig versorgt wird. Alle Aktivitäten, die mit einer Fenster-Maximierung zusammenhängen, sollen in unserem aktuellen Fall ausgelassen werden, nur die Berechnung der tatsächlichen Werte der Datenliste darf durchgeführt werden. Warum? Weil sonst die Sonderfunktionen im maximierten Fenster-Zustand nicht mehr vorhanden sind, denn ohne weitere Arbeit weiß ich dann nicht, ob das Fenster maximiert wurde oder nicht.

Das heißt also, dass die Listenbearbeitung nach dem Aufruf der internen Methode **Stretch** ergänzt werden muss, der bisherige Abfrageblock muss also vollständig in Abhängigkeit diesers Parameters erfolgen:

w_Navigator, $ChangeGraphic

```
If p_NotOnlyStretch
    If pEventCode='evMaximized'
```

Es gibt jedoch auch eine neue Verarbeitung im anderen Falle, anschließend an den bisherigen Abfrageblock:

w_Navigator, $ChangeGraphic

```
Else
    Calculate iListGraph.$cols.2.$name as l_Col2Name
    Calculate iListGraph.$cols.3.$name as l_Col3Name
End If
Calculate #F as $cinst.$redraw()
```

Der letzte Befehl **$redraw** existierte schon, er schließt die Routine wie seit eh und je ab. Wozu die Spaltenbenennung mit den beiden lokalen Variablen gut sein soll? Im Falle, dass nur der Dehnungsfaktor aus- oder angeschaltet wird, weiß ich ja nicht mehr ohne weiteres, ob das Fenster maximiert ist oder nicht, ich umgehe schließlich genau aus diesem Grund die regelmäßige Bearbeitung der optischen Gestaltung der Datenliste. Doch durch die Art der Erstellung dieser Datenliste, die immer einen **$define** beinhaltet, gehen bei einer erforderlichen Bearbeitung auch meine Spaltenüberschriften verloren. Deshalb merke ich mir ganz am Anfang der Fenster-Methode **$ChangeGraphic**, vor jeglichem sonstigen Tun, die bestehenden Spaltenüberschriften:

w_Navigator, $ChangeGraphic

```
Calculate l_Col2Name as iListGraph.$cols.2.$name
Calculate l_Col3Name as iListGraph.$cols.3.$name
```

Doch zurück zu unserer **$DoRightMouse**. Die Farben selbst lasse ich hier in dieser Methode gar nicht ändern, die Absicht freilich muss ich mir merken und den **Push Button Color** auftauchen lassen, damit Farben überhaupt ausgewählt werden können.

Probieren Sie diesen Code ruhig aus - er klappt trotz seiner erfrischenden Kürze prächtig, auch wenn ich mit den beiden Graphiken „Pie“ und „Scatter“ noch lange nicht zufrieden bin. Ich möchte jedoch zuerst den Rest der Tätigkeiten anpacken, die die **$DoRightMouse** erledigen soll.

Mein nächster Stolperstein ist der Verlauf. Bei den Anzeigeformen konnte ich schön die Konstanten benutzen, die mir Omnis im CATALOG unter „Constants“ bei den „Graph-Types“ bietet, doch bei dem Attribut für die Verlaufsform **$washdir** weigert es sich, die entsprechenden Konstanten bei „Graph-General“ anzunehmen. Mein Objekt **iObjGraph** sieht jedoch bis auf die Texte und ihre Positionen gleich aus. Also verdächtige ich nacheinander die Speicherverwaltung meines Betriebssystems, die eventuell überholte Version der externen **Wash Control**-Komponente oder sonstiger EXTERNALS, bis ich bei Pontius Pilatus angekommen bin - das Übliche halt, das Zeit schluckt und graue Haare macht. Dabei lerne ich noch die verschiedenen Fehlermeldungen in der nützlichen HASH #ERRTEXT kennen, ohne jedoch in diesem Umfeld zu verstehen, was sie bedeuten, bis ich endlich einsehe, dass mein Objekt **iObjGraph** eben doch nicht dasselbe ist wie das im EXAMPLE - zumindest nicht immer. Denn manchmal klappt doch alles prima!

Wo der Unterschied lag? Ich besorgte mir das Objekt per Name, das EXAMPLE und meine Graphik-Komponente hingegen per Event **evGraphObjChange**. Deshalb akzeptiert das Objekt manchmal die **Wash Control**-Parameter - und manchmal nicht. Ich hatte mich schon gewundert, warum sporadisch das Attribut **$areaeffect** wie

gewöhnlich über die Variablenaufbereitung der Rechten Maus oder über Programmcode zu ändern war und manchmal laut Omnis nicht einmal wirklich existierte. Woher ich Letzteres weiß? Nun, die Fehlermeldung **#ERRTEXT** behauptete stur und steif, die Notation wäre schlichter Blödsinn, ja es gäbe nicht mal eine Instanz und meine **$attributes**-Liste **#L1** zeigte mir das Attribut **$areaeffect** für **iObjGraph** gar nicht an, verkündete sogar, ein Toolbar-Objekt anzuzeigen. Hat im Übrigen viel Ähnlichkeit mit der Attribut-Auswahl, die die Methoden-Objekte meiner Datei-Tasks erzeugen.

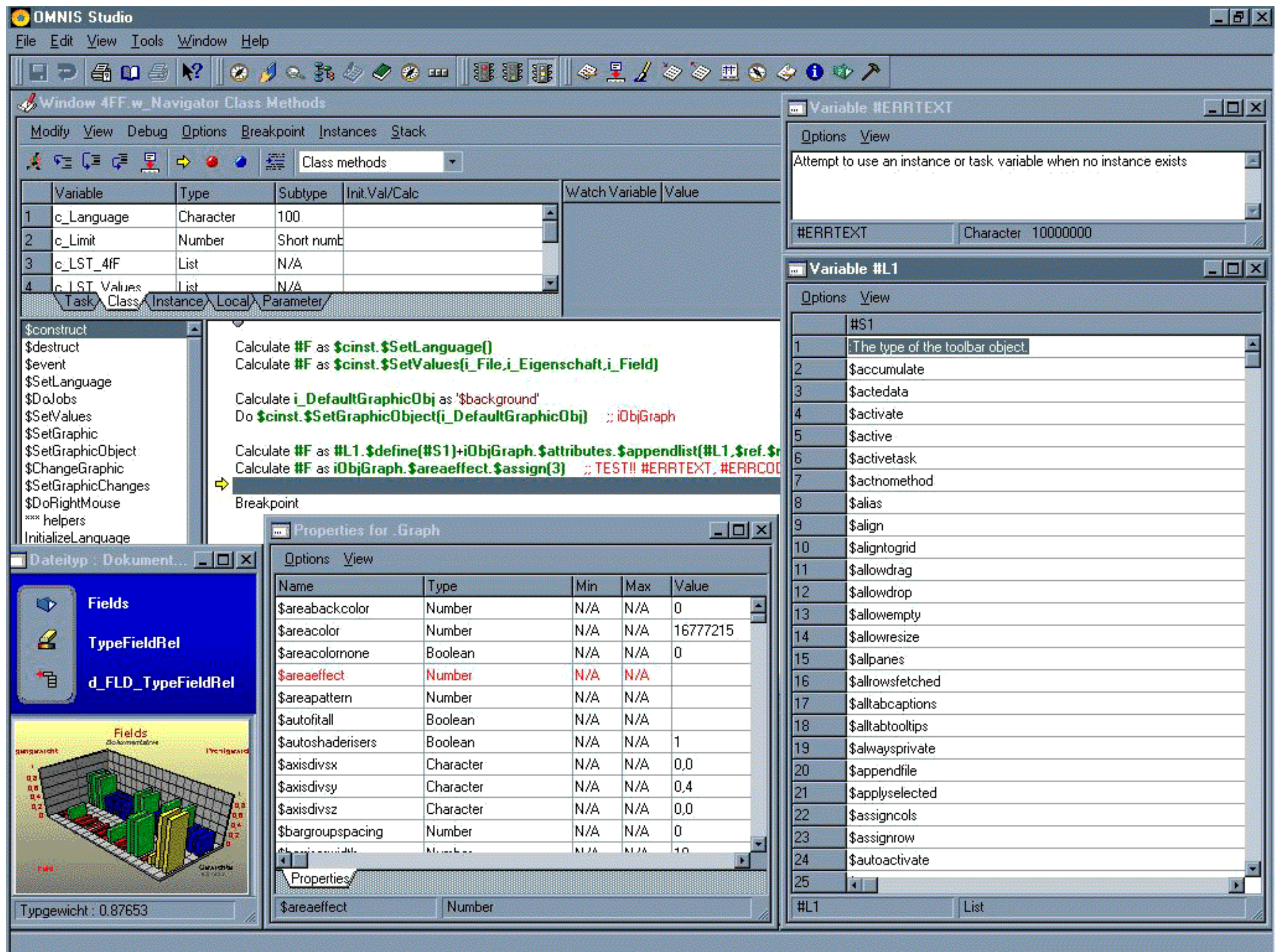

Abbildung 248 Fehlersuche

Da ich nun mit allen Tricks **$areaeffect** nicht dazu bringe, den Wert „3“ bzw. **kGRwash** anzunehmen und dies aber nun exakt die Bereitwilligkeit des Objektes bestimmt, Verläufe und verschiedene Farbwerte überhaupt anzunehmen, erkenne ich diese Verweigerung als das Kernproblem. Wieso ich trotzdem über die Methode **$cinst.$objs.File.$getobject** ein so passabel aussehendes Objekt zurückkriege und nicht bloß Leere? Weiß ich nicht, null Ahnung.

Ich finde mich aber damit ab und mache erst einmal tabula rasa, weg mit dem ganzen Zeug, das nicht funktioniert. Das bedeutet, dass ich aus der Fenster-Methode **$construct** den Aufruf zur Erstellung des Hintergrundobjektes ersatzlos streiche:

w_Navigator, $construct

```
Do $cinst.$SetGraphicObject(i_DefaultGraphicObj)
```

Danach fliegt der Parameter **p_ObjectName** aus der Fenster-Methode **$SetGraphicObject** heraus sowie die beiden Zugriffe **$cinst.$objs.File.$getobject**, die nicht mit dem Event **evGraphObjChange** funktionieren. Der ganze Anfang wird also gnadenlos eleminiert, wie die Abbildung unten zeigt.

Ok - das hat das Fenster gut verkraftet, funktioniert dennoch alles, was vorher funktionierte.

Leider ist mein Problem nicht gelöst, doch wenigstens weiß ich nun, woran es liegt, auch wenn mir Omnis natürlich nicht dieselben schönen Möglichkeiten für die externe Graphik-Komponente bietet wie für die eigenen Objekte und Funktionen. Zumindest, ja zumindest tut es das eben nicht ohne klare Kenntnis aller Möglichkeiten, also ohne das Graphic Manual, Doch bedeutsam genug, mir die Mühe der Beschaffung zu machen, ist mir diese Fragestellung denn auch wieder nicht. Deshalb bleiben mir nun zwei Möglichkeiten: ich deaktiviere die Menü-Zeilen der Popup-Menüs im Falle, dass die Anwender nicht das Hintergrund-Objekt angeklickt und damit den benötigten Event ausgelöst hatten oder ich gebe bei Bedarf einen Hinweis für die Anwender aus, dass sie doch bitte erst das Hintergrundobjekt aktivieren sollten.

Ich mache ersteres, möchte meinen Benutzern nicht unnütz auf die Finger klopfen. Außerdem füge ich ein weiteres Feld in meiner Statuszeile hinzu, auch dieses mit der Einstellung „sizing" = **kElastic**. In dieses Feld werde ich nun die verschiedenen Typen von Graphik-Objekten, die ich separat behandle, einstellen. Deshalb benötige ich fünf weitere Textfelder, denn neben diesen vier Typen will ich auch den Fall anzeigen anzeigen, dass kein Objekt ausgewählt wurde. Für solche unwandelbaren Bezeichnungen ist meine Textliste ganz praktisch.

In meine Fenster-Methode **$construct** füge ich also am Ende ein:

w_Navigator, $construct

Calculate #F as $cinst.$statusbar.$panes.2.$text.$assign(i_LST_Text(2,23)) ;; kein Objekt ausgewählt / no object selected

Dabei enthält bei meiner aktuellen Fensterkonfiguration Zeile 23 den Text, dass kein Objekt ausgewählt worden ist.

Für jeden meiner unterschiedlich behandelten Fälle gebe ich nun seinen Typ in der Statuszeile aus, aktiviere den **Push Button Color** und nach Bedarf die Menüzeilen.

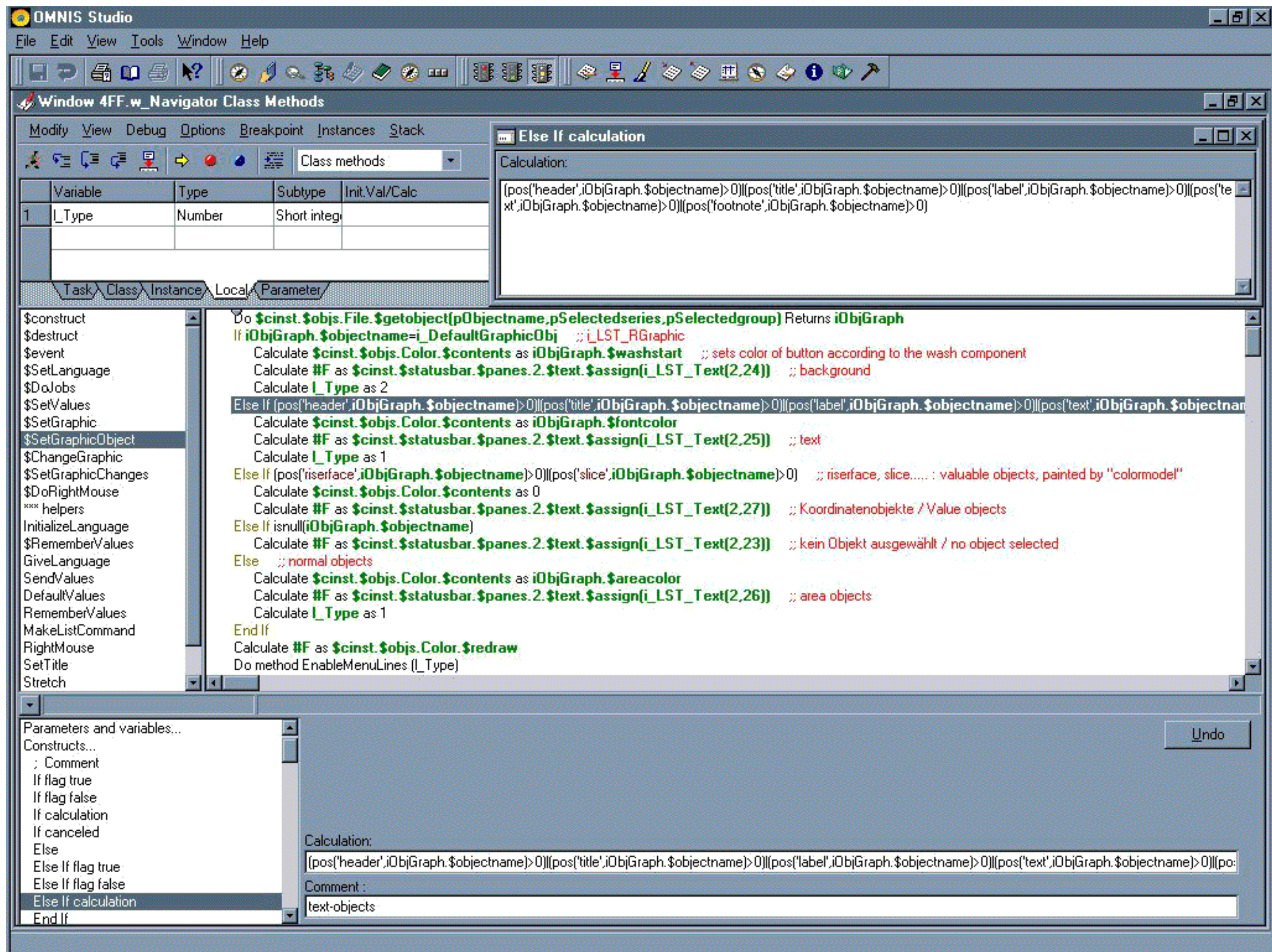

Abbildung 249 Fenster-Methode $SetGraphicObject

Für die Menüzeilen habe ich in der internen Fenster-Methode **AddGraphicChanges** noch eine lokale Variable **l_Type**, **short integer**, eingefügt, die ich zur Erkennung dieser Typen verwenden will. Deshalb lautet das allererste Statement dieser **AddGraphicChanges** nun:

w_Navigator, AddGraphicChanges

Calculate #F as i_LST_RGraphic.$define(i_Text,l_Command,l_Key,l_IconId,l_Type) ;; i_LST_RGraphic - Catalog, Constants == Graph-Types

Der initiale Wert dieser Variablen, Null, soll Unberührbarkeit bedeuten, das gilt für alle Fälle, die objektunabhängig sind, beispielsweise das Ändern der Anzeigeform oder der Wände. Solche Listenzeilen der Rechten-Maus - Liste **i_LST_RGraphic** sollen in jedem Fall zu aktiven Menüzeilen führen.

Das erste Mal, dass dieser Wert abweicht, ist beim Schlüssel „Color“, also muss ich dort zum ersten Mal auch einfügen:

w_Navigator, AddGraphicChanges

Calculate l_Type as 1 ;; color font/area

Der nächste Schlüssel, „ColorFirst“, erhält dann die zusätzliche Kennzeichnung „2“:

w_Navigator, AddGraphicChanges

Calculate l_Type as 2 ;; background

Sonst tue ich nichts in dieser Methode, kein weiteres Statement wird ergänzt, das heißt, dass alle folgenden Zeilen, die in dieser kleinen Routine zusammengefügt werden, denselben Wert „2“ erhalten, weil L_Type nicht mehr geändert wird.

Übrigens - damit die hübschen Graphiken keinen falschen Eindruck hinterlassen, sortiere ich in der Fenster-Methode **$SetGraphic** die Dateifelder noch nach ihren Eigenwerten, anschließend nach den Profilgewichten, schließlich ist die „räumliche“ Anordnung von Dateifeldern innerhalb der Datei-Datenstruktur nicht von Bedeutung und sollte damit nicht die Graphik beeinflussen.

w_Navigator, $SetGraphic

Calculate #F as i_LST_Graph.$sort($ref.C2,,$ref.C3)

Außerdem muss diese neue Routine zur Aktivierung bzw. Deaktivierung von Menüzeilen nun auch in dieser Fenster-Methode **$SetGraphic** geschehen, ganz am Ende:

w_Navigator, $SetGraphic

Do method EnableMenuLines (0)

Objektorientiert gehört diese Methode zwar in das Menü selbst, doch da ich dieses Menü schon die ganze Zeit vom Fenster anstatt von sich selbst, von „innen“ heraus, befehlige, spare ich mir hier die zusätzliche Übergabe von Steuerelementen an das Menü.
Wie diese neue Routine aussieht? Sie selektiert alle Zeilen, die kleiner oder gleich dem Parameter sind und deaktiviert die nicht selektierten. Also werden alle Funktionen mit dem Aktivitätstyp Null immer aktiviert. Nach den aktuellen Einstellungen der **$SetGraphicObject** werden deshalb die Hintergrundveränderungen nur für Hintergrundobjekte durchgeführt, Schriften und normale Objekte mit Flächenfarben erlauben nur einfache Farbenänderungen und die graphischen Darstellungen der gewünschten Werte gar keine, da sie nach einem Farbmodell nach ihren jeweiligen Werten und nicht nach ihrem Objekttypus koloriert werden.

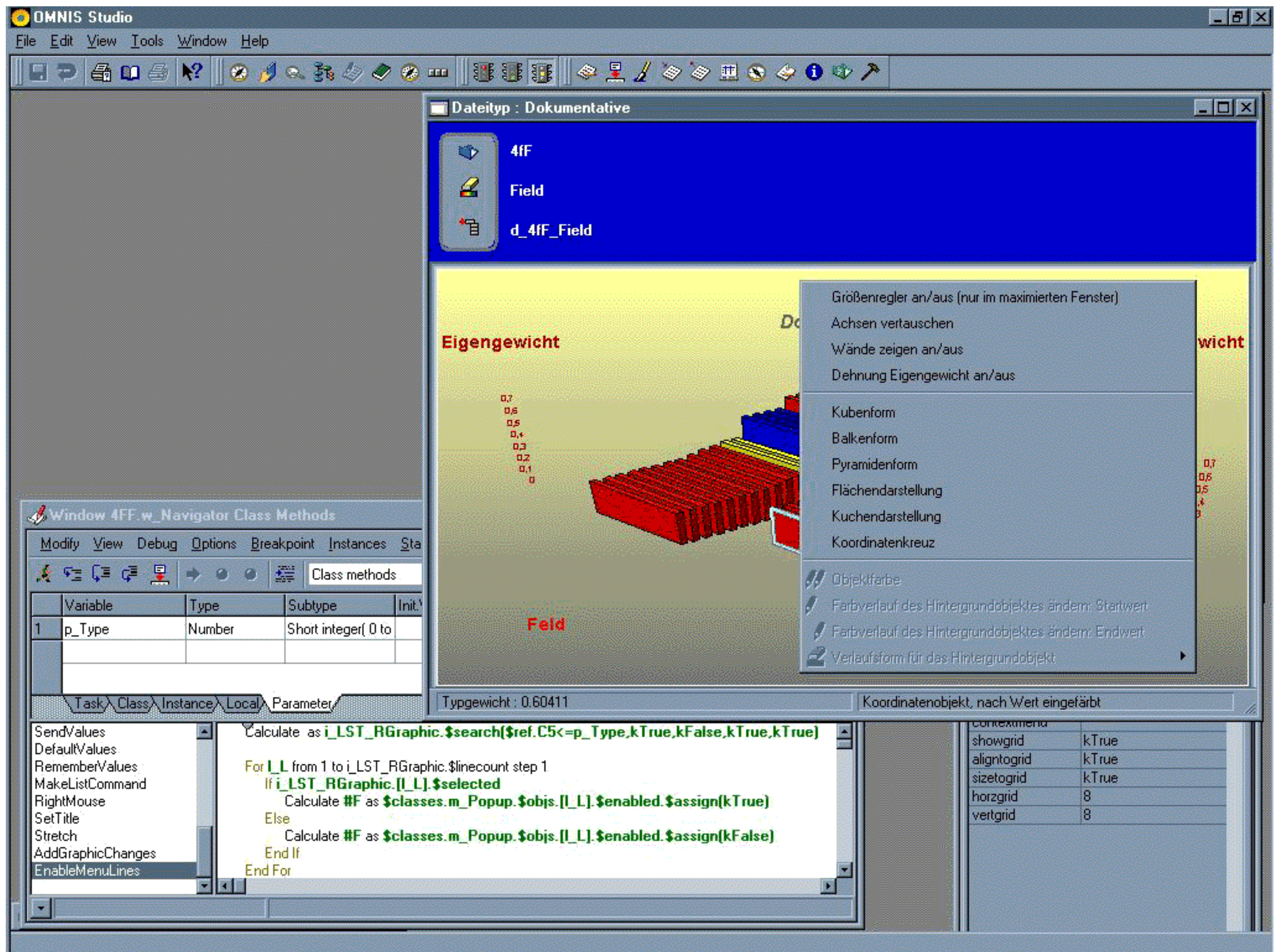

Abbildung 250 Interne Fenster-Methode EnableMenuLine

Da die nichtaktivierten Menüzeilen nicht ausgewählt werden können, muss ich mir gar keine Gedanken mehr machen, ob eine Auswahl gültig ist. Omnis erlaubt keine Auswahl von deaktivierten Menüzeilen. Deshalb schrumpft die Verlaufsbearbeitung für das Hintergrund-Objekt zu einem Fünf-Statement-Portiönchen zusammen, wobei die Prüfung des Attributs **$areaeffect** nur darauf beruht, dass natürlich der Hintergrund als gleichbleibende Fläche ausgemalt werden darf. Deshalb muss bei einer Verlaufsauswahl eben sicher gestellt sein, dass auch ein Verlauf vorliegt, denn eine gleichbleibende Fläche gilt verständlicherweise nicht als solcher.