



**Developer Manual
Version 3.2.8**

PlanBoard for Omnis Studio
© 2009 Master Object Consultancy

Copyright Notice & License

This manual is subject to the PlanBoard Developer License Agreement. The software may be used or copied only in accordance with the terms of the agreement. Resale is not permitted. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher or copyright owner.

Copyright © 2009 by Master Object Consultancy, The Netherlands. All rights reserved. Registered and unregistered trademarks mentioned in this manual are the property of their respective holders and have been used for informational purposes only.

Master Object Consultancy has tried to make the information contained in this manual as accurate and reliable as possible, but assumes no responsibility for errors or omissions. Master Object Consultancy disclaims any warranty of any kind, whether express or implied, as to any matter whatsoever relating to this manual, including without limitation the merchantability or fitness for any particular purpose. In no event shall Master Object Consultancy be liable for any indirect, special, incidental, or consequential damages arising out of purchase or use of this manual or the information contained herein.

For licensing information, please refer to
<http://www.masterobject.com/planboard> or write to:



Reigerskamp 393
3607 HX Maarssen
The Netherlands
e-mail: info@masterobject.com



Table of Contents

[1] About PlanBoard	
1.1 Introduction	
1.1.1 History of PlanBoard.....	5
1.1.2 About this Manual	6
1.1.3 Version History	7
1.2 About PlanBoard	
1.2.1 Conceptual Overview.....	8
1.2.2 PlanBoard Terminology	9
1.2.3 Using a PlanBoard Window	13
1.3 Object-Oriented Design	
1.3.1 Studio vs. Classic.....	16
1.3.2 Design Patterns	19
[2] Implementing PlanBoard	
2.1 About the PlanBoard Libraries	
2.1.1 Developer Pack Contents	26
2.1.2 Preferences Folder	26
2.1.3 Copyright Info and Credits	27
2.1.4 MasterStudio and PlanBoard Classes.....	27
2.2 Installation	
2.2.1 The Omnis Environment.....	28
2.2.2 Copying Base Classes	30
2.2.3 Copying PlanBoard Classes.....	31
2.2.4 License Registration	33
2.2.5 Deployment	33
2.3 Initializing PlanBoard	
2.3.1 Opening the MasterStudio Root Context.....	34
2.3.2 Instantiating a PlanBoard Delegate Object.....	36
2.4 Managing a PlanBoard Window	
2.4.1 Opening the window.....	40
2.4.2 Closing the window	41
2.5 Managing PlanBoard Contents	
2.5.1 Delegate Object Variables	42
2.5.2 The Legend List	43
2.5.3 The Resources List	45
2.5.4 The Planning Slots List.....	46
2.5.5 Handling Drag & Drop Actions.....	47
2.5.6 Changing Scales & Time Grid.....	48
2.6 Changing the Look & Feel	
2.6.1 Creating Subclasses	49
2.6.2 PlanBoard Architecture	50
2.6.3 PlanBoard Window Attributes.....	53
2.6.4 Adding your own fields and buttons.....	54
2.6.5 Extending PlanBoard's context menus.....	56
2.7 Printing	
2.7.1 Controlling destination, page and job setup	59
2.7.2 Modifying the look of reports	59
2.7.3 Adding your own objects.....	60



[3]	Upgrading from Omnis Classic	
3.1.1	Converting the Classic Library to Studio	61
3.1.2	Renaming Old Variables	62
3.1.3	Moving methods to your PlanBoard delegate object	62
3.1.4	Translated Strings.....	66
[4]	Programming Conventions	
4.1	Naming Conventions	
4.1.1	Libraries, Classes, Variables: Significant part on the right.....	67
4.1.2	Abbreviations	68
4.1.3	Libraries	70
4.1.4	Classes.....	70
4.1.5	View Objects (on Windows, Menus, Toolbars, Reports).....	74
4.1.6	Variables	75
4.1.7	Method Names	76
4.2	Coding Conventions	
4.2.1	Method Grouping and Ordering	78
4.2.2	Real Numbers.....	78
4.2.3	Reference Passing	79
4.2.4	Return Values.....	81
4.2.5	Reversible Blocks.....	81
4.2.6	Testing Object Instances.....	82
4.2.7	Using Parentheses when Calling Methods	83

[1]

About PlanBoard for Omnis Studio

1.1 Introduction

1.1.1 History of PlanBoard

PlanBoard for Omnis Classic

Thank you for purchasing PlanBoard! This manual helps you get the most out of PlanBoard for Omnis Studio. You'll have your data displayed in PlanBoard within a few hours. With a few more days of work, you may well be using PlanBoard as the primary means to create, edit and maintain your scheduling database...

PlanBoard started as an in-house project at VDA, a software house employing over 40 developers (many of them Omnis developers) in Hilversum, The Netherlands. Customers expressed a need for graphical visualization of planning and scheduling data. After spending some time researching available products such as Omnis externals and third-party Project Management tools, VDA found a unique way to get around their many restrictions. Most third-party options ran on one platform only, had a high per-project license cost and were difficult to implement and learn. In addition, being dependent on third parties for maintenance of a core part of a product is something to be avoided. Enter "unlocked source code".

By applying growing knowledge about object-oriented development, VDA spent about half a year on the first PlanBoard project. Although the Classic version was not intended to be a product sold to other developers, there was a growing awareness of the widespread need for scheduling software. Therefore, VDA added a page to its web site and got in touch with several Omnis developers that quickly jumped at the chance to use PlanBoard. In its first three years, PlanBoard for Omnis Classic became a core part of many Omnis applications.

MasterStudio

Upon acquiring the rights to PlanBoard in 2000, Master Object Consultancy developed and released version 2.0 of PlanBoard for Omnis Classic. We also embarked on the development of an advanced object-oriented framework for Omnis Studio: MasterStudio.

MasterStudio is a framework written in Omnis Studio, consisting of several libraries, the classes of which form ready-to-use *framework components*, allowing quick development of advanced GUI applications. While MasterStudio was built entirely in Omnis, its main inspiration



PlanBoard for Omnis Studio

comes from other frameworks in the Java community. It thus brings proven object oriented concepts and additional power to Omnis Studio developers.

MasterStudio was designed from the start to provide a clean implementation of object oriented design patterns in Omnis Studio: For the development of MasterStudio, no Omnis Classic code was used whatsoever.

PlanBoard 3.0 for Omnis Studio, described in this manual, was a complete redesign and rewrite of PlanBoard that was built on top of the MasterStudio framework's *foundation layer*. PlanBoard for Omnis Studio was designed so third-party developers can easily implement it without prior knowledge of the MasterStudio framework.

Master Object Consultancy specializes in developing Omnis frameworks. We are a *D2D* (Developer to Developer) company. We develop components and frameworks exclusively for use by other Omnis developers. We trust you'll enjoy the results of our efforts!

1.1.2 About this Manual

Sections

This manual assumes that you know how to develop Omnis Studio applications and that you understand basic object-oriented design such as working with multiple object instances, subclassing and inheritance.

Section 1 introduces PlanBoard and explains some object-oriented design theory that we used in the MasterStudio framework. PlanBoard itself uses object-oriented techniques such as subclassing and polymorphism extensively. Chapter 1.3 explains the most important OO concepts used for PlanBoard development.

Section 2 is a systematic guide on how to create a working PlanBoard window, starting at installation of the Developer Pack and ending with deployment.

Section 3 describes the programming style and naming conventions used by PlanBoard and other frameworks built on top of the MasterStudio foundation layer.

Screen Shots

Examples and screen shots used throughout this manual were copied from the supplied demo application. We strongly recommend that you look at the source code as it contains many additional comments.

Color Coding

In this manual, variable names, constants, method names, messages, attributes and other literals that belong to PlanBoard or Omnis have been coded **blue**. Other literals (such as database field names) belong to your own application and you can rename or change them at will.

1.1.3 Version History

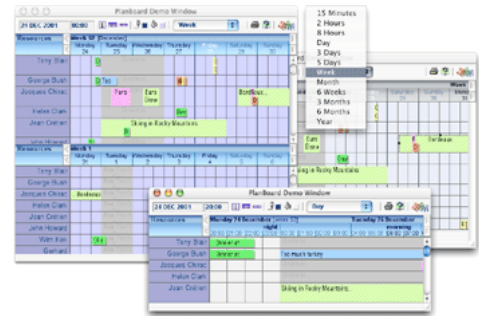
- This chapter lists enhancements made after release 3.0 of this Developer Manual. Please refer to <http://www.masterobject.com/planboard> for the very latest release notes, including bug fixes and known issues.
- Version 3.2.8** The manual improves terminology and incorporates minor changes to reflect the last planned release of PlanBoard 3. PlanBoard 3 is supported in Omnis Studio 4.1 through 4.3. It is based on the MasterStudio 1.1.0 framework foundation layer.
- ⇒ *PlanBoard is no longer supported on Omnis Studio 3 or 4.0. The upcoming PlanBoard release 4.0 will support Omnis Studio 4.3.1 and will also work with Omnis Studio 5.0.*
- Version 3.2.5** The manual was updated to reflect the move to Master Object Consultancy and the latest features that were added since version 3.1.1. Also, screen shots were added to reflect the new Studio Browser in Omnis Studio 4. See the Master Object Consultancy website for detailed release notes.
- Version 3.1.1** This release fixed a number of minor bugs and syncs PlanBoard with the latest version of MasterStudio, including numerous enhancements in its base framework layer. PlanBoard-specific enhancements include:
- Improvements were made to layout and terminology used in this manual.
 - To better comply to Microsoft and Apple font usage guidelines, field styles `MoPbResource` and `MoPbButtonbar` now use the *Tahoma* font on Windows and *Lucida Grande* on Mac OS X.
- Version 3.1.0** Apart from many bug fixes and performance enhancements, version 3.1 of PlanBoard added the following features:
- The `$slotOverlap` attribute of the PlanBoard delegate object allows developer control of the overlap or stacking of planning slots in a PlanBoard pane. This is described in chapter 2.5.4.
 - The `$addContextCommand` allows you to add your own commands to PlanBoard's context menus. See chapter 2.6.5.

1.2 About PlanBoard

1.2.1 Conceptual Overview

PlanBoard is a collection of Omnis Studio classes (a *framework*) that allows visualization and management of time slots in Gantt chart style windows. Typical applications include appointment scheduling, employee planning, project planning, event calendaring, broadcast channel programming, car rental, room rental, resource allocation, machine use planning, advertisement scheduling, frequency allocation and ticket sales.

What makes PlanBoard especially powerful is that it cleanly separates the user interface (which visualizes the schedule and allows users to change it) from the database (which PlanBoard knows nothing about -- that's your responsibility). From the start, PlanBoard was developed to be as flexible as possible to you, the developer. Therefore, it allows you to flexibly change content and terminology to be displayed. PlanBoard cleanly hides the technical details of the user interface so you don't need to worry about its complexity, unless you want to.



To you, PlanBoard's classes are encapsulated by a single object per window instance. This object is called the *PlanBoard Delegate* and is further explained in chapter 1.3.2. You control a PlanBoard window by subclassing this object and changing some of its custom methods and attributes. To further enhance a PlanBoard window you can add your own objects to the window class.

PlanBoard receives its data from your delegate object using Omnis lists. You build these Omnis lists yourself, using any database mechanism that you like. For optimum performance, PlanBoard retains a reference (pointer) to your planning slots and resources lists (see chapter 1.2.1). So there's only one copy of the data in memory.

You only fetch records from your database that are needed during the currently selected time frame. If a PlanBoard window is currently showing the first week of January, you'll only fetch planning slots that are in that period. Likewise, you should only send resources to PlanBoard that the user can actually see. If you fetch more resources from the database everything will still work, but PlanBoard becomes slower updating and redrawing data that the user cannot see anyway.

When data in your database changes, you simply tell PlanBoard to redraw itself after you rebuild your corresponding list. PlanBoard reads new values from your list (to which you've given PlanBoard a

reference). In return, when the user modifies data displayed in PlanBoard, the delegate object allows you to store the changes from the list to the database.

PlanBoard was entirely written in Omnis Studio and its classes are supplied mostly unlocked. Therefore, you can make changes and enhancements to the product. You can do this by subclassing framework classes so you can easily install future updates from Master Object Consultancy without modifying any of your own code. In addition, if you encounter a bug and know a solution before we or other developers do, you can fix it without waiting for us to come out with an update.

1.2.2 PlanBoard Terminology

What all PlanBoard applications have in common is that they help users plan *something* (or someone) at a certain *time* with a certain *goal*. To help you understand PlanBoard, we'll first introduce terminology used for each of these. Of course, you can change the terminology and prompts your end users get to see.

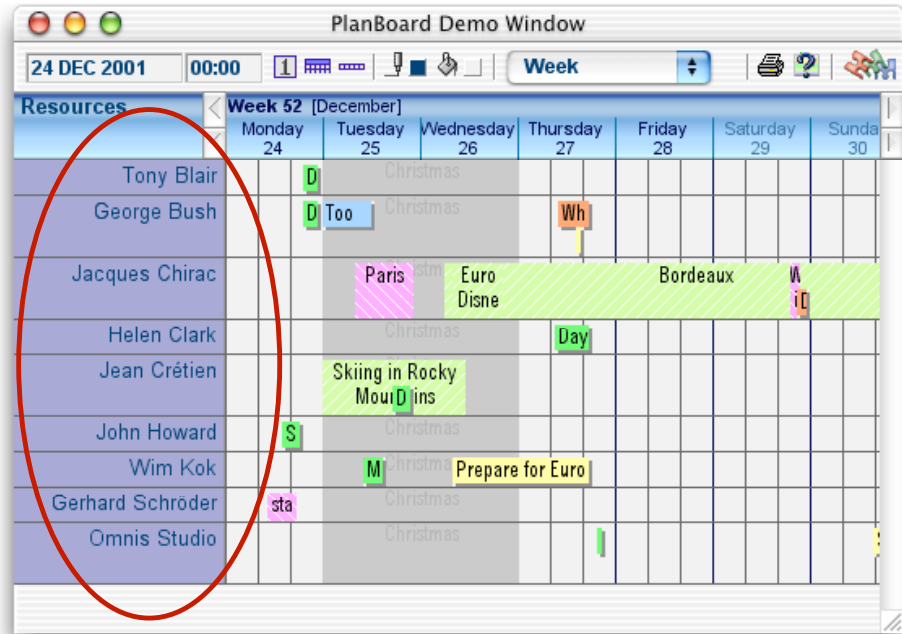
In this manual, we use the term *PlanBoard* mostly to refer to the complete developer product you purchased. However, end users typically refer to *PlanBoard* as a single window instance. To avoid confusion, we'll use *PlanBoard window* to refer to a window instance.

PlanBoard allows you to open multiple windows simultaneously, each window showing different data, perhaps of a completely different nature. In a hospital management application, one window could be used to schedule operating rooms, whereas another window is used to schedule doctor appointments. In PlanBoard, these windows are subclasses of the `PbWindowSuperclass` window. You can easily add your own objects to a PlanBoard window.

Note that screen shots printed here are based on the default layout, `kPbLayoutHorizontal`, where *things* to be planned are at the left and *time* is at the top. Future versions of PlanBoard may support different layouts, where *things* could be at the top and *time* could be at the left.

Resources

In PlanBoard, we call whatever is being planned (the *something*) a *resource*. Because resources are to be planned, they can be available or unavailable at any time. Your users do not typically use the word *resource*, so you will replace it by a more descriptive term such as "employee", "machine", "room", etc. In the demo application provided, resources are people from the *employees* table. In this manual and throughout the PlanBoard framework, we call them *resources*.



Resources

Resource Text

Each resource has a name or description that is drawn in the resource header of PlanBoard. If you have text stored in an existing database field, you can have PlanBoard use the corresponding column from your resources list. If you don't have a separate database field for resource text, then PlanBoard automatically adds a text column and allows you to calculate it after fetching records from the database. This is especially handy if you want to concatenate each person's first and last names.

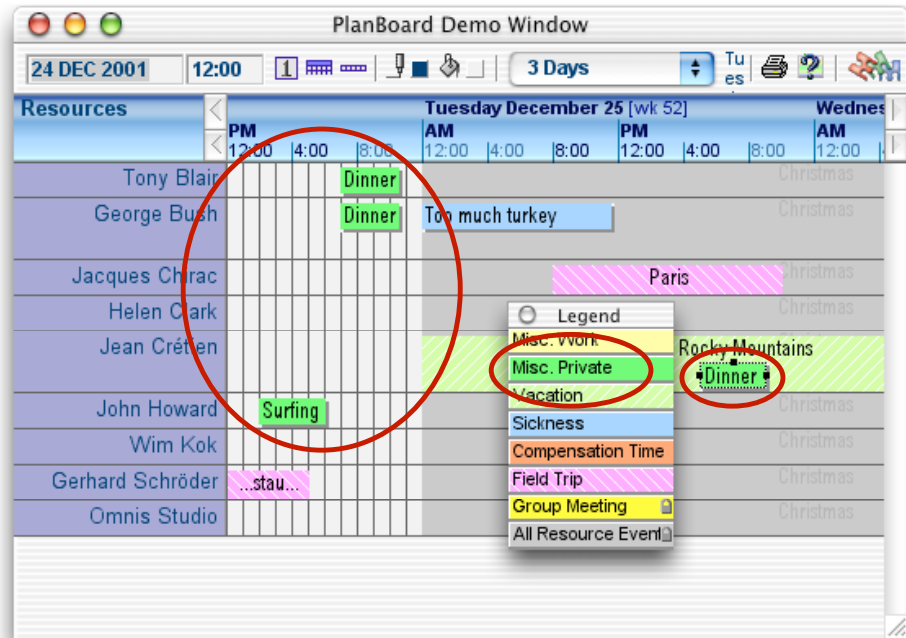
Planning Slots

Once something is planned at a certain time with a certain goal, we call it a *planning slot*. Each planning slot has a starting time and an ending time. An individual planning slot derives its color from the *goal* for which it was planned. A planning slot appears on the line or column that corresponds to its resource.

Slot Groups

Typically, many planning slots have the same *goal* and thus have the same color when displayed by PlanBoard. This is why we use the term *planning slot group* or simply *slot group* to indicate the goal for which something was planned. Each *slot group* has a corresponding entry in the color legend. So the color legend is a list of *slot groups* with associated information.

In the demo application provided, each slot group represents a reason for a person to be planned, such as "vacation" or "meeting".



Slots belonging to the same group have the same color derived from the legend

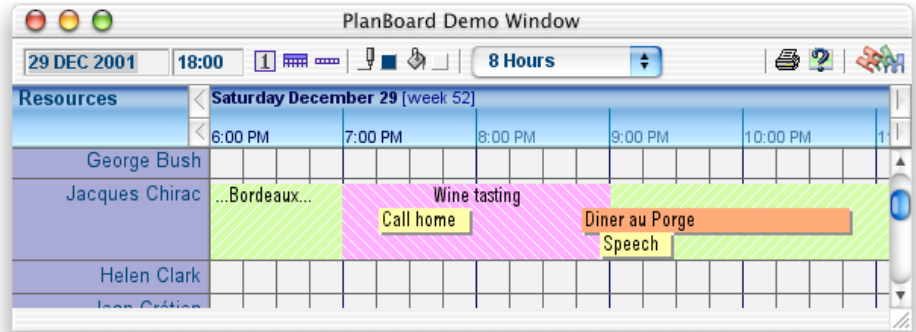
Legend Privilege

Each slot group displayed in the color legend has a Boolean value that determines whether users can create new slots by dragging from the legend into PlanBoard. Your delegate object sets this Boolean value during the instantiation of a PlanBoard window. In the picture above, the last two legend rows display a lock indicating that they cannot be used to create new slots.

Slot Types

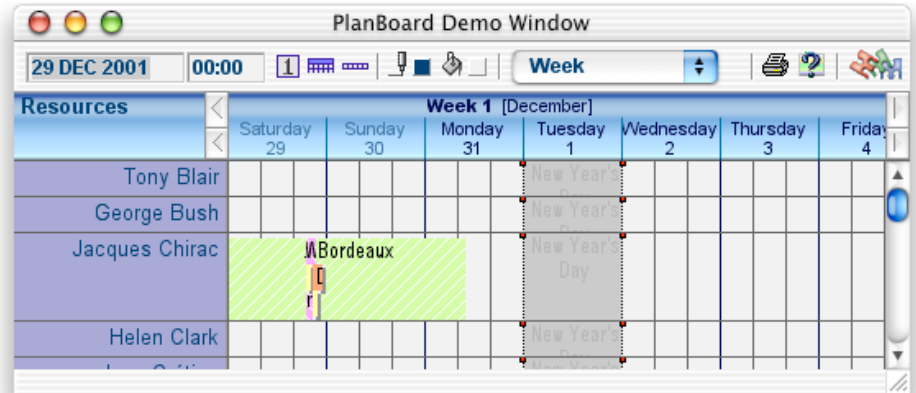
In PlanBoard, planning slots have one of three available types. *Normal* planning slots belong to a single resource and are displayed below or to the right of other planning slots for the same resource.

Private background planning slots also belong to a single resource (thus *private*), but they are drawn behind any other planning slots of the same resource (thus *background*). Private background slots are useful for long events (such as an employee’s vacation) that do not cause conflicts with other events (such as a dinner appointment). A private background slot is also useful for a person’s birthday.



Private background (“Bordeaux” and “Wine tasting”) and normal planning slots

Public background planning slots belong to all resources (thus *public*) and are drawn behind (and slightly above or to the left of) other planning slots. Public background slots are typically used for public holidays, weekends, or any other regular times that are valid for all resources.



Public background slot (“New Year’s Day”)

To help distinguish background slots from foreground slots that may have a similar (or even the same) color, you can optionally give background slots a pattern. Since background slots are always drawn behind other slots, you may not see them if there are other background slots in front of them.

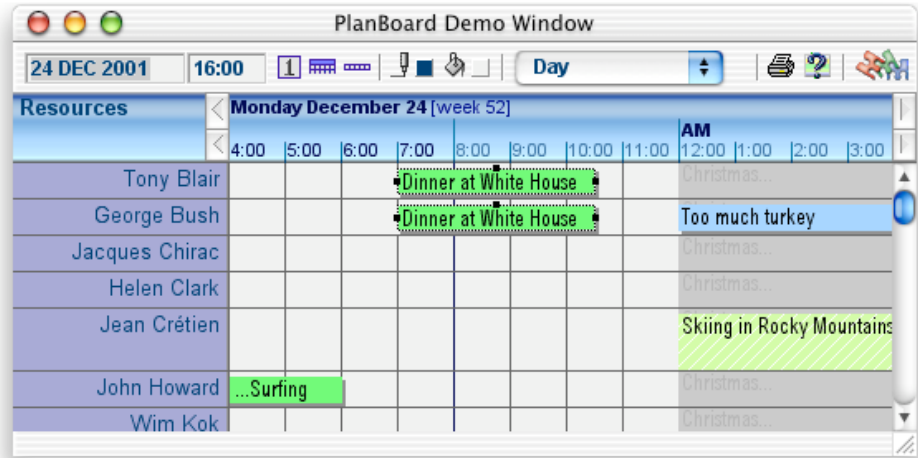
Slot Text

Each planning slot has its own descriptive text (which is drawn on the planning slot and appears on its tooltip). If you have this text stored in an existing database field, you can have PlanBoard use that column of your planning slots list. If you don’t have a database field for the slot’s text, then PlanBoard automatically adds a text column and allow you to calculate it after fetching records from the database.

Linked Slots

Normally, when a user drags a slot, only that single planning slot in the database is updated. PlanBoard also allows you to link planning slots so that they are always moved simultaneously. You do this by including a *link column* of type long integer to your planning slots list. When a user selects a planning slot that has a non-zero value in this field, PlanBoard

also selects all other objects with the same number. After the user drags the slots, it is up to your delegate object to update all linked records in the database (some linked slots may be in a different period and thus aren't in the slots list at that time).



Linked slots ("Dinner at White House")

Slot Privileges

Each planning slot has a number of user privileges associated with it, such as whether the slot may be deleted, whether it may be moved to a different time, whether it may be resized (shortened or lengthened), and whether it may be moved to a different resource. If you have these privileges stored as Boolean values in existing database fields, you can have PlanBoard use the corresponding columns of your planning slots list. If you don't have database fields for each slot's privileges, then PlanBoard automatically adds appropriate columns to your list and allows you to calculate them after fetching records from the database, unless you want them to always be `kTrue`.

Database Prerequisites

Your database needs at least a *resources table* and a *planning slots table* (or file classes if you use Omnis native data file commands). Each record in these tables must have a unique numeric key that may not be zero. If you don't have a unique numeric key, you could add one to your database or use a unique sequence number that you add to your lists before refreshing PlanBoard.

In database terms, a *planning slot* always has a foreign key to its *resource* and a foreign key to its *slot group*. The number of slot groups (corresponding to legend entries) is usually small and in many PlanBoard applications they are not stored in the database. In your delegate object, you can create a local list of slot groups and send that to PlanBoard to use for the legend. You can also build the legend list from a database table if you wish.

1.2.3 Using a PlanBoard Window

Time Navigation

The time line has two arrow buttons on each side. The lower button allows you to scroll in *minor increments* that correspond to the first colored divider line and the upper button scrolls in *major increments* that

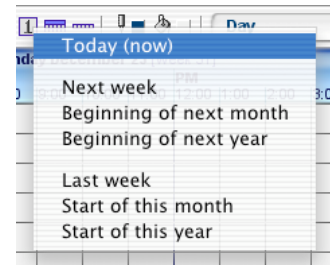


correspond to the currently selected scale. PlanBoard supports continuous scrolling. Simply keep the mouse down on any of the scrolling buttons and time will scroll by indefinitely. PlanBoard automatically redraws (and gets from your database) its planning slots after scrolling half a screen distance.

The arrow buttons have a tooltip that shows additional keyboard shortcuts. As long as the cursor isn't in a date entry field (press **Tab** or **Escape** to get out of the field) users can use the left and right arrow keys to scroll to a different time. Hold **Shift** with an arrow key to scroll in major increments.



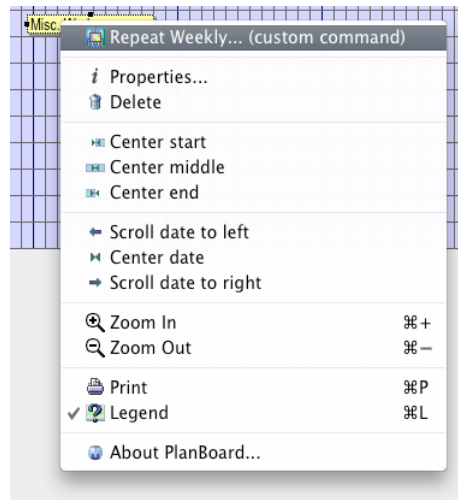
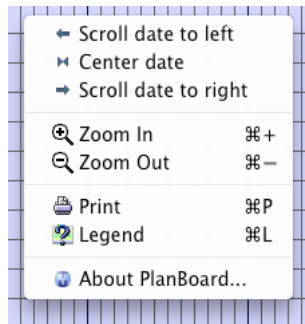
The default buttonbar includes two fields in which users set PlanBoard's starting date and -time. These fields can easily be accessed using the **Tab** key. A tooltip on these entry fields lists additional keyboard shortcuts that are used for date entry.



Additional buttons in the default buttonbar offer different ways to scroll to specific dates, to a specific month, or one to four weeks ahead or back. PlanBoard developers can easily add their own date navigation options, perhaps by adding a menu or a floating window (palette).

Context Menus

Timeline and PlanBoard grid support right-mouse clicking to quickly move to a different date. You can either center the date clicked on the screen, move it to the left margin, or move it to the right. After right clicking on a planning slot, the user may center the planning slot's begin date, end date or middle on the screen.



Context menus (after right-clicking on the background and on a planning slot)

Color Schemes

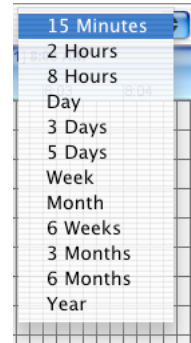
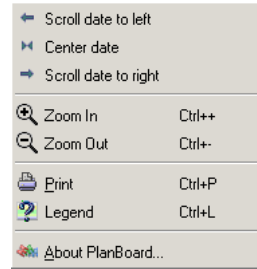
Users choose from four color schemes, each supporting seven different color or gray backgrounds. PlanBoard remembers the currently selected user settings per PlanBoard window instance. Developers can open



multiple window instances, each using a their own color scheme, window title and terminology.

Time Scales

PlanBoard supports twelve time scales ranging from 15 minutes up to a full year (based on a 1024x768 sized display). Each time scale has its own button bar and vertical divisions. The default buttonbar has a popup list allowing users to switch to a different scale.

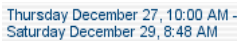


By single clicking in the timeline, users can select a different time scale and center it around the clicked date. By right clicking anywhere in the PlanBoard window, the user can zoom in or out to the next time scale using the context menu.

Each scale has a developer-adjustable time grid to which planning slots automatically “snap” during drag & drop operations.

PlanBoard developers can add their own ways of selecting scales, perhaps by adding menu lines or toolbar buttons. Developers can also easily limit scale options that are available to the user.

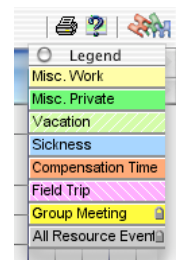
Status Information



The default buttonbar has a status area to the right of the scales popup list. This area shows a description of the current action being performed. It also shows planning slot details (such as the exact beginning and ending date) as the user holds the mouse over specific planning slots and while the user performs drag & drop operations.

Legend Palette

The legend palette is used to create new planning slots. The palette is opened using a dedicated button in the default buttonbar or by using the **Control-L** keyboard shortcut (**Command-L** on the Mac). The palette closes automatically as another window becomes active and it re-opens when PlanBoard comes back to the front.



Creating planning slots

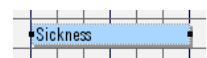
Dragging the desired slot group from the legend into the PlanBoard grid creates planning slots. During creation of a Planning slot, the cursor changes into a plus sign to reflect the current user action.

After the user clicks in a valid region of PlanBoard to set the starting time for the planning slot, a black line indicates the length of the newly created slot. By clicking, the planning slot is created with its current length, indicated in the status area of the default buttonbar.

A drag & drop operation can be canceled by moving the cursor out of PlanBoard and releasing the mouse, or by releasing the mouse while it's over an invalid part of the PlanBoard window.

Resizing planning slots

Depending on user privileges under control of the PlanBoard developer, users resize planning slots by grabbing one of their resize handles after clicking on the slot. The drag





Switching slots to a different resource

handles are black if the slot can be resized. They are gray if the slot is locked in its current position. The drag handles have a red center if the slot cannot be resized. In that case, the slot can only be moved to a different time in its entirety (without changing its length).

Depending on user privileges under control of the PlanBoard developer, planning slots can be moved to a different resource. Slots have a third dedicated drag handle at their center that allows users to do this.



Moving planning slots

To move a planning slot to a different time, users simply drag after single clicking to select the slot. PlanBoard scrolls automatically as the user approaches the edge of the PlanBoard grid. PlanBoard uses two auto-scrolling speeds, depending on how close the cursor is to the edge of the PlanBoard grid.

Duplicating planning slots

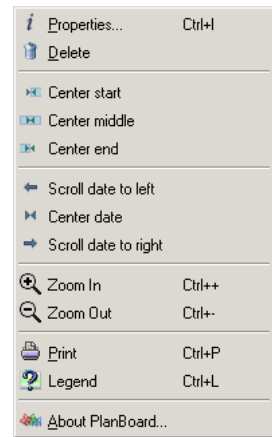
Users can duplicate a planning slot (as long as the user has create privileges for the corresponding slot group) by option dragging (Mac) or control-dragging (Windows) it to a different location.

Deleting planning slots

Under control of developer and user-specific privileges, planning slots are deleted by selecting them and pressing **Delete** (or **Command-Backspace** on the Mac).

Getting planning slot info

Developers can create their own “popup window” that appears when a user types **Control-i** (or **Command-i** on the Mac) after selecting a planning slot.



Users can right-click a planning slot to retrieve its properties, to delete it, or to position it on the screen. A context menu also includes options that correspond to the date on which the user right-clicked the mouse.

Getting slot and legend info

PlanBoard allows developers to open their own window after the user single-clicks on the name of a resource or on an item in the legend palette.

1.3 Object-Oriented Design

This chapter contains optional background information that helps you understand the architecture of PlanBoard. If you want to get started with PlanBoard quickly, you can skip to section - and come back to this chapter later.

1.3.1 Studio vs. Classic

Before PlanBoard for Omnis Studio, PlanBoard for Omnis Classic had a proven record of accomplishment. The Classic version of PlanBoard became a success largely because it used object-orientated design in the

non-object-oriented world of Omnis Classic. Here's a description of some important object-oriented features and the way we moved them to the Studio version of PlanBoard.

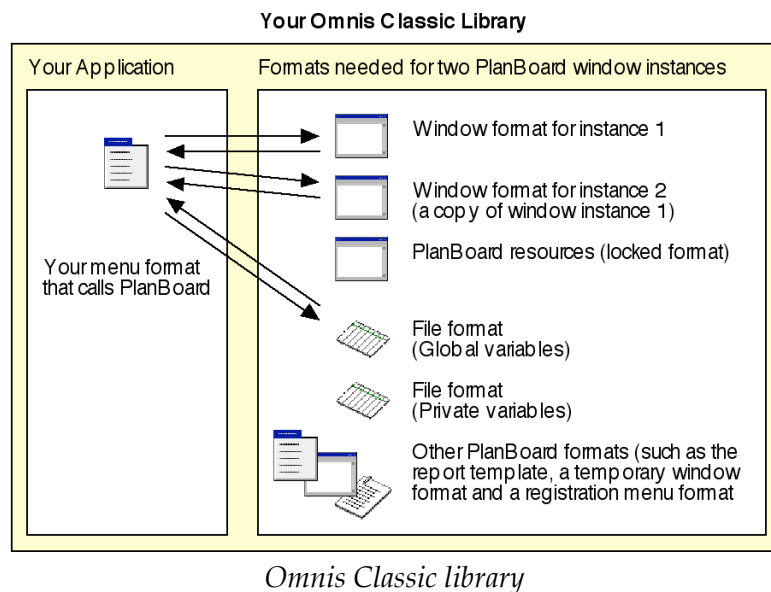
Encapsulation

By separating PlanBoard's functions from the rest of an application, its internal complexity is hidden from the PlanBoard developer. You "talk" to PlanBoard by using its *interface*: The public methods you call and the *callback methods* (and *custom attributes*) that you provide for PlanBoard to call.

In Omnis Classic, we designed our own standards to enforce encapsulation. The PlanBoard developer had to learn which methods he or she was allowed to call, and which methods he or she wasn't allowed to call. The distinction between *protected*, *private* and *public* methods was made with strict naming conventions and the use of a central dispatcher procedure in each format to call its internal (private) methods. To allow multiple PlanBoard instances to exist simultaneously, most PlanBoard code was included behind the window format itself, working with a large number of format variables. To implement PlanBoard in their own application, developers provided their own menu format that implemented predefined *callback methods*.

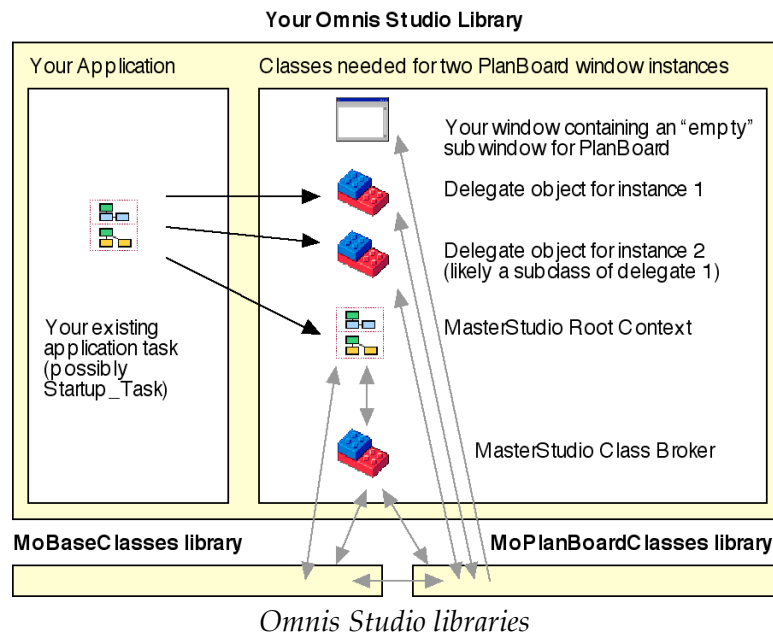
In all, there was a strict dependency on the way PlanBoard worked, looked, and communicated with your application. The Classic version worked very well, but interdependencies limited flexibility: If we wanted to extend or change things internally, we were likely to break existing PlanBoard applications. In addition, there was no clear distinction between the developer's PlanBoard code and the rest of the application.

The following picture shows formats needed in your main library to implement two PlanBoard windows in an Omnis Classic application:



The Studio version of PlanBoard makes the programming interface considerably easier by providing an object superclass for your PlanBoard instances. This superclass object, of which you create subclasses called *PlanBoard delegate objects*, implements the complete interface that is needed between your application and PlanBoard. The actual work of PlanBoard is carried out by many other classes (tasks, windows, controller objects etc.) that are hidden from the developer in a separate library (if you want, you can still move those classes into your central library for deployment).

The internal structure of PlanBoard for Omnis Studio can be extended easily, without breaking dependencies on your application. In the following picture, black arrows indicate method calls that you provide. Gray arrows indicate dependencies that are maintained automatically.



Contexts and *delegation* are further explained in chapter 1.3.2.

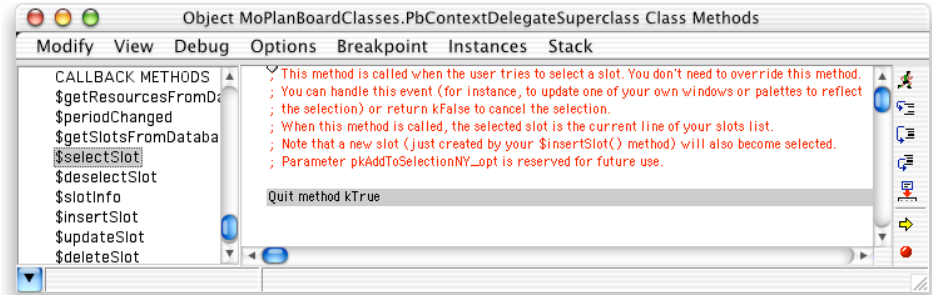
Unicity

PlanBoard works well in a large range of applications, largely because each method implements just one simple, well-defined function. PlanBoard offers just those functions that you need to display and graphically manipulate your data, but it doesn't do anything that would create a dependency on the database or on other global aspects of your application. PlanBoard offers well-defined methods, each of which has one function that can be easily understood.

In the Studio version, these methods are cleanly separated into the PlanBoard delegate object. Some *callback methods* in the Classic version did more than just one thing. For example, PlanBoard called the `PB_SLOT_MODIFICATION` method both after a new planning slot was created, after an existing slot was modified, and after a slot was deleted. In the Studio version of PlanBoard, each function has its own dedicated *delegate method*. Most of these methods have a default implementation in



the *delegate superclass*. Therefore, you'll only override those methods and custom attributes that you need in your application.



Some delegate methods in the PlanBoard delegate superclass

Overriding and Inheritance

The Classic version of PlanBoard only allowed a limited number of functions to be changed by the developer, although some developers have taken advantage of source code provided to “patch” internal procedures behind the Classic PlanBoard window format. The Studio version supports subclassing of PlanBoard objects, which means that it is no longer necessary to change any internal code in order to add your own functionality. PlanBoard accesses its classes through a central *class broker* object. If the class broker finds a class with the same name as one supplied in a Master Object Consultancy library, PlanBoard automatically uses the class that has the highest version number. PlanBoard also uses an *abstract class factory object* that gives you even more freedom in using your own subwindows.

Note that for most development work, you don't need to subclass any of PlanBoard's internal objects. The *delegation design pattern* (described in chapter 1.3.2 below) offers an easy way to extend PlanBoard without needing any knowledge of its internal architecture.

1.3.2 Design Patterns

PlanBoard and the MasterStudio framework make extensive use of object-oriented features such as encapsulation, subclassing and polymorphism. It is beyond the scope of this manual to explain these features in detail and you don't need to be an expert in them to successfully implement PlanBoard. PlanBoard for Omnis Studio was designed so it can be implemented without knowing anything about the way it works internally.

However, it is useful to note that the architecture of PlanBoard and other Master Object Consultancy frameworks was largely based on existing solutions and design principles, often referred to as *design patterns*. In this chapter, we'll give you a brief description of some of the design patterns we used in PlanBoard.

Although books and web-based materials about design patterns are mostly based on different programming languages (such as Smalltalk, Objective C, C++ and Java), we have found that knowledge represented by design patterns is also very useful for Omnis Studio development.



Model-View-Controller (MVC)

Master Object Consultancy certainly did not write the book on design patterns, and we are well aware that much of what we are saying here is a practical simplification of what design patterns are all about. If you would like to know more about design patterns, we recommend reading the book "Design Patterns - Elements of Reusable Object-Oriented Software" by the Gang of Four (ISBN 0201633612).

Design patterns offer a way to document reoccurring software problems. Design patterns document how to solve problems in an efficient and future-proof way. Although mapping design patterns to Omnis Studio classes isn't always trivial, you will find that design patterns help build solutions that make your software easier to enhance and reuse. By using design patterns, you invest in knowledge and understanding of other software systems. Moreover, if you use a recognized pattern to solve a problem, your code should be easier to understand and maintain by other developers.

One of the great strengths of object-oriented development is that it allows us to apply *real world modeling*, where objects in our programming language match objects that exist in the real world. In case of PlanBoard, we built objects that represent the individual elements of a PlanBoard window. You will create objects that represent the data in the database. Those objects are used to *feed* the PlanBoard window with information.

The *Model-View-Controller* design pattern (MVC in short) helps developers create well-maintainable applications by cleanly dividing the responsibilities of the application into separate logical objects. MVC originated in the early years of object-oriented software development. The Smalltalk programming language (most notably its built-in object browser) made extensive use of it. Later, the NeXTstep application kit was largely based on MVC. It later evolved into Apple's Cocoa Application Kit and WebObjects Java Client frameworks. Most modern frameworks, many of them Java-based, use the MVC design pattern in some way. Many parts of Omnis Studio also implement MVC, even though much of it is hidden from the Omnis developer by the 4GL language and high-level classes that often combine the *controller* with either a *model* or a *view*.

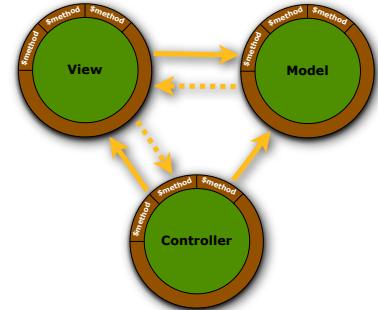
One aspect of well-maintainable applications is that it should be easy to fix bugs when they occur. MVC helps us move toward that goal because it guides us in determining which objects perform each specific task. In MVC applications, every object gets a clear responsibility to be either part of the *model*, the *view*, or the *controller*. When a bug is detected, it is often more obvious which object could have been the cause of it. Moreover, you can almost be certain that a specific function is only located in one place, effectively achieving *single point of definition* (SPOD).

Perhaps an even bigger advantage of the MVC pattern is that it promotes reuse of *model* and *view* objects. The architects of the MVC design pattern realized that *model* and *view* usually carry out responsibilities that are not application-specific. Well-designed *model objects* and *view objects* can be reused in different applications without

any changes. The MVC pattern implies that *controller objects* may very well be the only part of an application that is application-specific.

Once we have cleanly separated out the *model*, *view* and *controller*, we can start thinking about creating fully functioning windows that work completely independently from the underlying database. We should be able to simply replace the model by a different one, and our views still work, perhaps by changing a line or two of code in the controller. We can also easily switch to a different view without any changes in the model.

Of course, the above goals can be achieved in many ways. The MVC design pattern guides us by not only telling us the goal, but also showing us how to achieve it. Most importantly, it helps us decide how a controller manages the model and the view. Without going into too much detail, we'll summarize the most important steps that a controller takes to connect the model(s) with the view(s). These steps correspond to the arrows in the picture on the right.



- *Controller connects to model*
Before any object can exist, you need to instantiate at least one controller that is responsible for starting up an application or for constructing an application component. Once the *root controller* is instantiated, it connects to a model. The model can be any object that represents data to be managed by an application or component: It could be data stored in a SQL database. It could just as well be text stored in a flat file or even XML data retrieved from the Internet.
- *Controller creates a view*
In order to do anything meaningful, an application or component needs at least one view. A view is anything that represents data from the model. It could be a window. It could also be a popup menu. Or it could be a report or an XML export file. Once a view is instantiated, the controller tells it which model to use. This implies that a view doesn't know anything about the model until its controller gives it that information. The controller obviously has a choice of which view to use. And different controllers connecting it to different models could use the same view.
- *View interrogates controller*
Before a view is of any use, it needs a reference to the model it should use. It gets that information from its controller. If a view needs its own separate database connection, it might ask its controller to create one. Once the view knows the model it should use, it very likely works independently from its controller until user actions take place that the controller handles.



- *View accesses model*
Once the view knows what model to use, it usually accesses the model without needing further intervention from its controller. It is important to realize that the model may be given some parameters by the view about how it should return its data (for instance, which fields from the database are needed by the view). However, the model should know nothing about the presentation of the data. That is the sole responsibility of the view, which may get some additional information (such as user preferences) from its controller.
- *Model returns data*
Once one or more views have registered an interest, a model supplies data to its views, and the model keeps track of changes in the data by those views.

MVC and Omnis Studio

One of the great things about MVC is that it allows views and models to be reused and combined in flexible ways. Omnis Studio goes a long way towards giving developers separation between the view and the model. *View objects* and *model objects* are mostly implemented as *window class instances* and *table class instances* (that you work with as Omnis lists).

As a 4GL, Omnis Studio makes it very easy to simply instantiate a window and have it construct a table (list). As a result, many Omnis Studio applications create strong links (dependencies) between the window, its menus or toolbars, and the database (session and table). There is nothing wrong with that for rapid development.

However, if you want to create reusable components it makes sense to remove the *controller* from the window, and implement it as an independent object class or task class. The controller then acts as a container of the model or a pointer to a central model. The controller can also keep track of its view's state. Using this technique, you can easily create "persistent" windows that open showing exactly the same data as before they were closed. Contrast this to a standard Omnis window instance that loses its instance variables (data and model) when closed.

MVC in PlanBoard

PlanBoard for Omnis Studio was developed with *model-view-controller* in mind. As a result, PlanBoard doesn't have any internal code that depends on the model used. PlanBoard works with any database as long as you tell its *controller* (through the PlanBoard delegate object) how to do it.

In addition, without changing PlanBoard's internals, you can subclass its views to enhance them with your own objects (see chapter 2.6). Each section of a PlanBoard window is a separate view object (implemented as an Omnis subwindow). Future versions of PlanBoard may even come with several layout options that work without any changes in the model.

Contexts and Controllers

As explained above, a view should not know very much at all about its environment. Instead, its controller controls a view. Typical applications require many views and thus you need many controllers. To organize



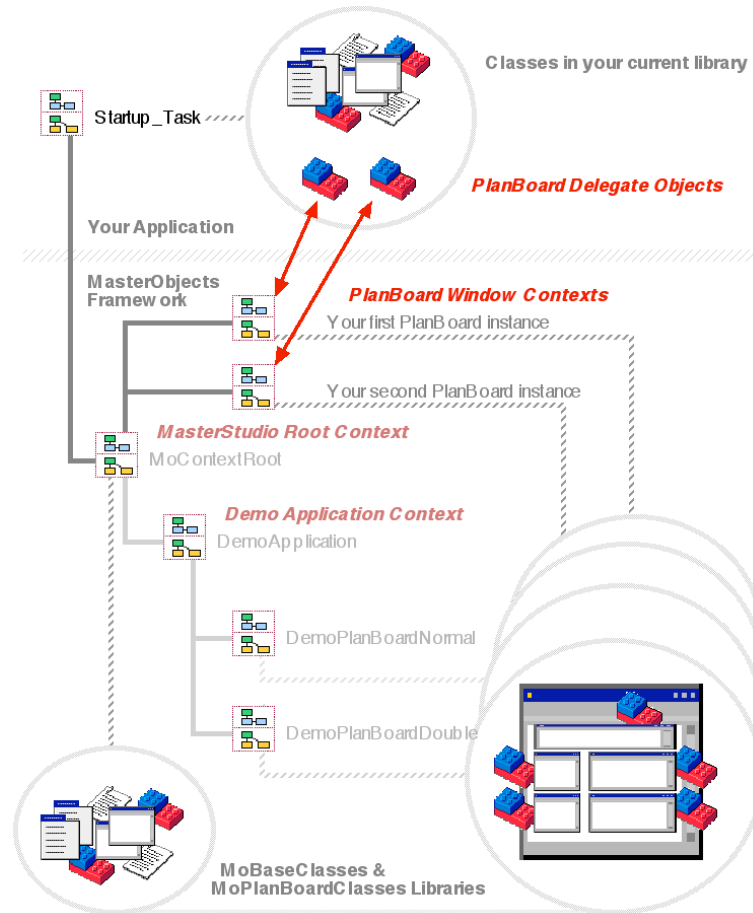
this plethora of objects, it makes sense to store them in a hierarchy: A tree-like structure where every branch of the tree is responsible for managing a small part of the application.

Model-View-Controller is an *aggregate design pattern*, which means that it actually consists of several design patterns that work together. One of these is the *Composite* design pattern, which organizes the views and controllers into a hierarchy as described in the previous paragraph. Typically, the root of the object tree is instantiated first: It provides services for all other objects to work with, and it knows how to instantiate those other objects. The root of the object tree is the *context* that all other objects live in.

Of course, the root of all Omnis Studio objects is Omnis itself: Omnis provides a runtime environment in which all of your objects are instantiated. By default, Omnis first instantiates the *startup task* of your library. Unless you open additional tasks, the startup task is the context in which all of your Omnis object instances live. Omnis even allows you to create *task variables* that are available to all of the objects that belong to the same task.

As mentioned above, Omnis allows you to do everything in a single task. But this contradicts the MVC and Composite design patterns: It might make more sense to divide an application into components, each component with its own hierarchy of objects that are cleanly *encapsulated* from the rest of your application. This is exactly what the MasterStudio does. By opening multiple contexts (implemented as Omnis tasks) for objects to live in, we cleanly divide the responsibilities of our applications into largely independent components.

The following figure shows the hierarchy of contexts (tasks) that MasterStudio instantiates, and how it connects to your application. Chapter 2.3.1 of this manual explains how your existing task opens the root of the MasterStudio controller hierarchy. This *root context* provides an environment for PlanBoard (and other framework applications, such as the demo application provided) to run in.



The figure above also shows that the demo application has its own task (the *demo application context*), and that two sample PlanBoard contexts are opened by the demo.

Note that Omnis Studio does not organize tasks hierarchically--they all show up under the `$itasks` node of the Notation Inspector. MasterStudio implements the *Composite* design pattern to instantiate Omnis tasks and Omnis objects as *contexts* and *controllers* in a *controller hierarchy*. In fact, a *context* is an Omnis task that implements a *controller interface*, just like its parent and children (including lists of *subcontroller* objects) do. In other words: If Omnis would allow tasks and objects to inherit from the same class, *context* and *subcontroller* would both be subclasses of *controller*.



Delegation

The *Delegation* design pattern (which, in PlanBoard and MasterStudio, is based on *template method*) offers a way to determine and influence an object's behavior, without the need to create a subclass.

Chapter 2.3.2 explains how you'll instantiate *delegate objects* that represent *PlanBoard instances* (windows) in your application. Each PlanBoard instance is a context (Omnis task instance) in which many subcontrollers and subwindows do their work. These internal objects are completely encapsulated by your delegate object. This means that your application does not need to access any of PlanBoard's objects directly, nor do you need any knowledge of PlanBoard's internal controller hierarchy (you can read more about PlanBoard's internal architecture in chapter 2.6.2).

PlanBoard is a complex component that uses many internal objects to do its work. Although it is possible to create subclasses of the PlanBoard context and any of its subcontrollers, this does require a lot of knowledge on your part. It would also make it difficult for Master Object Consultancy to change and enhance PlanBoard's internal methods and attributes without "breaking" your subclasses.

Following the Delegation design pattern, we have moved essential attributes and methods of the PlanBoard component into a separate object called the *PlanBoard delegate superclass*. This superclass contains only those methods and attributes that you are likely to need, making the delegate object very easy to use.

Other Design Patterns

Internally, PlanBoard uses several other design patterns. It is beyond the scope of this manual to describe them all. We do want to mention the *Abstract Class Factory* pattern, however. This pattern is very powerful, as it allows you to change PlanBoard's behavior by subclassing or even replacing its internal classes, *without having to change the PlanBoard context itself*. To create instances of its objects, PlanBoard uses the *context class factory*. By simply subclassing this object, you can have PlanBoard use your own classes instead of the default ones. This is explained further in chapter 2.6.1.