
IdeaBlade
Cabana Reference Application
Architecture and Class Documentation

William C. Jensen
June, 2007

Table of Contents

TABLE OF CONTENTS.....	3
TABLE OF FIGURES.....	5
1 INTRODUCTION.....	6
1.1 OVERVIEW	6
1.2 DEFINITIONS	6
1.3 ARCHITECTURAL LAYERS	6
1.3.1 <i>Microsoft-Supplied Components</i>	6
1.3.2 <i>Third Party Components</i>	7
1.3.3 <i>IdeaBlade Components</i>	7
1.3.4 <i>User Developed Components</i>	7
2 MICROSOFT COMPOSITE UI APPLICATION BLOCK (CAB)	8
2.1 OVERVIEW	8
2.2 CAB APPLICATION LEVEL.....	8
2.2.1 <i>Structure</i>	8
2.2.2 <i>Classes</i>	9
2.3 CAB WORKITEM LEVEL.....	10
2.3.1 <i>Structure</i>	11
2.3.2 <i>Classes</i>	11
2.4 CAB MODULE LEVEL.....	12
2.4.1 <i>Structure</i>	12
2.4.2 <i>Classes</i>	12
3 IDEABLADE/CAB	13
3.1 GLOBAL DEFINITIONS	13
3.1.1 <i>Global Name Definitions</i>	13
3.2 VIEWS	13
3.2.1 <i>Common View/Presenter/Context Support</i>	14
3.2.2 <i>Control Suite Specific View Support</i>	17
3.2.3 <i>ViewFactoryService</i>	17
3.2.4 <i>Control Views</i>	19
3.2.5 <i>Grid Views</i>	21
3.2.6 <i>Control Suite-Specific Grid Views</i>	23
3.2.7 <i>Task Navigation Views</i>	25
3.2.8 <i>Control Suite Specific Navigation Views</i>	27
3.2.9 <i>Default Navigation Implementation Selection</i>	28
3.2.10 <i>NavBarService</i>	28
3.2.11 <i>General LayoutViews</i>	29
3.2.11.1 <i>SearchPageView</i>	30
3.2.11.2 <i>SearchAndResultsPage</i>	30
3.2.11.3 <i>MasterDetailView</i>	30
3.2.11.4 <i>PageView</i>	31
3.2.11.5 <i>SummaryDetailView</i>	31
3.2.11.6 <i>EditorView</i>	32
3.2.12 <i>Shell Layout Views</i>	32
3.2.12.1 <i>Developer Express Shell Layout View</i>	34
3.2.12.2 <i>Dot Net Shell Layout View</i>	35
3.3 GRID BUILDERS	36
3.3.1 <i>Common Grid Builder Classes</i>	36
3.3.2 <i>Grid Builder Services</i>	37
3.3.3 <i>Control Suite Specific Grid Builder Classes</i>	38

3.3.4	<i>Default Grid Builder Implementation Selection</i>	40
3.3.5	<i>Tab View Controllers</i>	40
3.4	WORKSPACES	42
3.4.1	<i>CABWindowWorkspace</i>	42
3.4.2	<i>WindowWorkspace</i>	42
3.5	SERVICES	42
3.6	WORKITEM CONTROLLERS.....	42
3.6.1	<i>Entity View Controllers</i>	42
3.7	ENTITY MANAGEMENT	44
4	SKELETON APPLICATION	47
4.1	APPLICATION SHELL	47
4.2	CONFIGURED SERVICES	48
4.2.1	<i>DevForce Services</i>	48
4.2.2	<i>SCSF Services</i>	48
5	THE CABANA REFERENCE APPLICATION	48
5.1	COMMON INTERFACE ELEMENTS	49
5.1.1	<i>Name Constant Definitions</i>	49
5.1.2	<i>Interfaces</i>	50
5.1.2.1	<i>IOrderEditorController</i>	50
5.1.2.2	<i>IVerifierResultsAlertsView</i>	50
5.2	COMMON INFRASTRUCTURE COMPONENTS	50
5.2.1	<i>Services</i>	50
5.2.2	<i>Grid Builders</i>	50
5.2.3	<i>Tab View Controllers</i>	51
5.2.4	<i>Menu Extensions</i>	51
5.2.5	<i>Verifiers Setup</i>	51
5.2.6	<i>Entity Model Base Views</i>	51
5.3	CABANA FOUNDATION MODULE	52
5.3.1	<i>IdeaBlade/CAB Services</i>	52
5.3.2	<i>Cabana-Specific Services</i>	53
5.3.3	<i>Shell Display Setup</i>	53
5.3.4	<i>Navigation Bar Setup</i>	53
5.3.5	<i>Command Handlers</i>	53
5.4	CABANA TASK MODULES	54
5.4.1	<i>Splash</i>	54
5.4.2	<i>Admin</i>	54
5.4.3	<i>SharedEditor</i>	54
5.4.4	<i>SalesRep</i>	54
5.4.5	<i>SalesRepNav</i>	54
5.4.6	<i>SalesOrder</i>	54
A1	APPENDIX 1: GLOBAL DEFINITIONS	54
A1.1	GLOBAL NAME DEFINITIONS	55

Table of Figures

FIGURE 1:	CAB APPLICATION LEVEL CLASS STRUCTURE.....	9
FIGURE 2:	CAB WORKITEM CLASS STRUCTURE.....	11
FIGURE 3:	COMMON VIEW/PRESENTER/CONTEXT SUPPORT CLASS STRUCTURE	15
FIGURE 4:	CONTROL SUITE-SPECIFIC VIEW SUPPORT CLASS STRUCTURE	18
FIGURE 5:	CONTROL VIEW SUPPORT CLASS STRUCTURE	20
FIGURE 6:	COMMON GRID VIEW SUPPORT CLASS STRUCTURE	22
FIGURE 7:	CONTROL SUITE-SPECIFIC GRID VIEW SUPPORT CLASS STRUCTURE	24
FIGURE 8:	TASK NAVIGATION VIEW GENERAL SUPPORT CLASS STRUCTURE	26
FIGURE 9:	TASK NAVIGATION VIEW CONTROL SUITE SPECIFIC CLASS STRUCTURE	27
FIGURE 10:	SELECTION OF PRIMARY CONTROL SUITE-SPECIFIC TASKNAVIGATION IMPLEMENTATION.....	28
FIGURE 11:	GENERAL LAYOUT VIEW CLASS STRUCTURE	30
FIGURE 12:	DEVELOPER EXPRESS SHELL LAYOUT VIEW CLASS STRUCTURE	34
FIGURE 13:	DOT NET SHELL LAYOUT VIEW CLASS STRUCTURE.....	35
FIGURE 14:	COMMON GRID BUILDER CLASS STRUCTURE	37
FIGURE 15:	CONTROL SUITE SPECIFIC GRID BUILDER CLASS STRUCTURE.....	39
FIGURE 16:	SELECTION OF PRIMARY CONTROL SUITE SPECIFIC GRIDBUILDERBASE IMPLEMENTATION	40
FIGURE 17:	TAB VIEW CONTROLLER CLASS STRUCTURE	41
FIGURE 18:	ENTITY VIEW CONTROLLERS CLASS STRUCTURE.....	43
FIGURE 19:	ENTITY MANAGER CLASS STRUCTURE	46
FIGURE 20:	SKELETON APPLICATION CLASS STRUCTURE	47

1 Introduction

1.1 Overview

IdeaBlade/CAB is IdeaBlade's collection of tools and techniques for creating rich, enterprise-wide applications using the DevForce framework. Based upon the Composite UI Application Block (CAB) created and supplied by Microsoft's Patterns and Practices group, the IdeaBlade/CAB extensions provide an extensible architectural framework in which multiple teams of architects and developers can implement, deploy and support complex enterprise applications. The architecture and components support multiple third-party control suites, allowing applications to adhere to corporate-wide look-and-feel standards. Future versions of IdeaBlade/CAB will support user interfaces based on the Windows Presentation Foundation from Microsoft. The IdeaBlade/CAB architecture provides designated sites for centralized implementation of company and application-wide business rules and logic.

The IdeaBlade/CAB framework also includes standard implementations of many commonly-used components; shell layouts, layout views, data views and services. These allow rapid creation of prototype and pilot applications. As an organization's knowledge and skill levels increase, developers are free to create custom components to meet more advanced requirements.

1.2 Definitions

Term	Definition
CAB	The Composite UI Application Block developed and supplied by the Microsoft Patterns and Practices group.
SCSF	Smart Client Software Factory – a Microsoft-supplied tool for creating skeleton components based on the CAB framework.
IdeaBlade/CAB	IdeaBlade's extensions to the CAB framework that assist in creating full enterprise applications based on the DevForce application framework.
Cabana	A sample reference application

1.3 Architectural Layers

The IdeaBlade/CAB architecture comprises several distinct layers, embodied in assemblies referenced by application projects:

1.3.1 Microsoft-Supplied Components

CAB

The basic composite UI framework is supplied by Microsoft in several assemblies. These are used in unmodified form by the IdeaBlade/CAB extensions.

Smart Client Software Factory (SCSF)

Microsoft supplies a tool for creating skeleton application components based on the CAB libraries. While valuable, this tool is complex and leaves the developer with significant tasks to integrate an application with IdeaBlade DevForce or third-party control suites. IdeaBlade used the SCSF to create some of the IdeaBlade/CAB components and thus the SCSF

libraries must be referenced by applications based on IdeaBlade/CAB, but developers do not interact directly with the SCSF tools.

1.3.2 Third Party Components

Third-Party Control Suites

CAB and the IdeaBlade/CAB extensions natively support the Windows Forms (WinForms) control set supplied by the .NET framework. In addition, IdeaBlade/CAB supports two popular third-party control suites: DeveloperExpress and Infragistics. Full WinForms designer support is available for all control suites.

Microsoft is transitioning its primary UI rendering technology to the Windows Presentation Foundation (WPF). A limited WPF control suite is currently supplied by Microsoft as part of WPF (Dot Net release 3.0) and third-party control suites will be available in the future. Future releases of the IdeaBlade/CAB extensions will seamlessly support WPF-based views and selected control suites.

1.3.3 IdeaBlade Components

DevForce

A chief strength of IdeaBlade/CAB is its integration with the DevForce framework. The services, views and view builders of IdeaBlade/CAB are tailored to work with business entity classes created by DevForce. However, the DevForce product and its generated classes remain indifferent to the application and user interface technology; the model libraries generated by DevForce may be used without modification in applications that are or are not based on IdeaBlade/CAB.

IdeaBlade/CAB

The IdeaBlade/CAB extensions comprise a group of libraries (DLLs) and Visual Studio templates for creating application-level projects. These define interfaces, services, views, modules and other CAB components either for use directly in an application or as base classes for custom implementations.

IdeaBlade/CAB supplies individual libraries dedicated to each of the supported control suites (WinForms, DevEx, Infragistics). Application projects reference one or more of these based on the technology(-ies) used by the application. If multiple technologies are used, the application architect chooses one as the “primary” technology to be used by new views by default. Views that do not use the default technology must explicitly inherit from the appropriate technology-specific base class.

1.3.4 User Developed Components

Application Shell

The application shell is the top-level executable assembly hosting the user’s application. IdeaBlade/CAB supplies a custom version that architects customize for a specific application.

Application Infrastructure Assemblies

Skeleton projects supplied by IdeaBlade/CAB allow architects and selected developers to create common components available to all developers. These may provide common view layouts, enforcement of business rules or other functions that must be maintained centrally.

Modules

Modules are separate assemblies that form the unit of deployment in a CAB application. The choice of whether to load a module into the application is made at runtime, but once loaded all components of a module are configured into the application.

Modules may depend on one another, but only through loose coupling mechanisms provided by the CAB framework.

Application Foundation Modules

An application may create one or more foundation modules (assemblies) that supply common infrastructure for the application such as the basic screen and navigation mechanisms seen by users.

Application Business Modules

Following the CAB architecture, application modules implement the bulk of an application's business functionality. With access to common facilities from the infrastructure assemblies, teams of developers can work independently on multiple parts of an application without interfering with one another.

2 Microsoft Composite UI Application Block (CAB)

2.1 Overview

Microsoft's Patterns and Practices group develops and promulgates architectural guidance and best practices for use of .NET technologies in real-world scenarios. Guidance often takes the form of an application block—a collection of classes, published in both library and source code form with documentation, that provides a starting place for application architects and developers. The Composite UI Application Block (CAB) is one such offering, comprising classes and structures that help create medium to large scale enterprise applications with enhanced scalability and maintainability. *[Add references/links to P&P and CAB]*

The central principle of the CAB architecture is *composition*; applications are composed at runtime by starting with a basic *shell*, then dynamically adding capabilities in the form of *modules*. Loose coupling between components is maintained at all times; provided facilities are named and made available through a hierarchy of well-known objects called *WorkItems*. Components locate and consume needed facilities by name and interface.

A full description of the CAB architecture is beyond the scope of this document, but the following sections provided a high-level overview of the structure of a CAB-based application.

2.2 CAB Application Level

2.2.1 Structure

At the highest level, CAB provides classes that establish the basic structure of a composite application. The application shell class is the entry point for the application; it supplies the static `Main()` method that instantiates and invokes the CAB application classes. This allows the application shell to inherit from the appropriate visual class, such as `System.Windows.Forms.Form`, for the UI rendering technology used by the application.

The application level defines two important classes:

- `Workspace` is the holder for visual elements.
- `WorkItem` is the element of the hierarchy that holds all facilities created and used by components of the application.

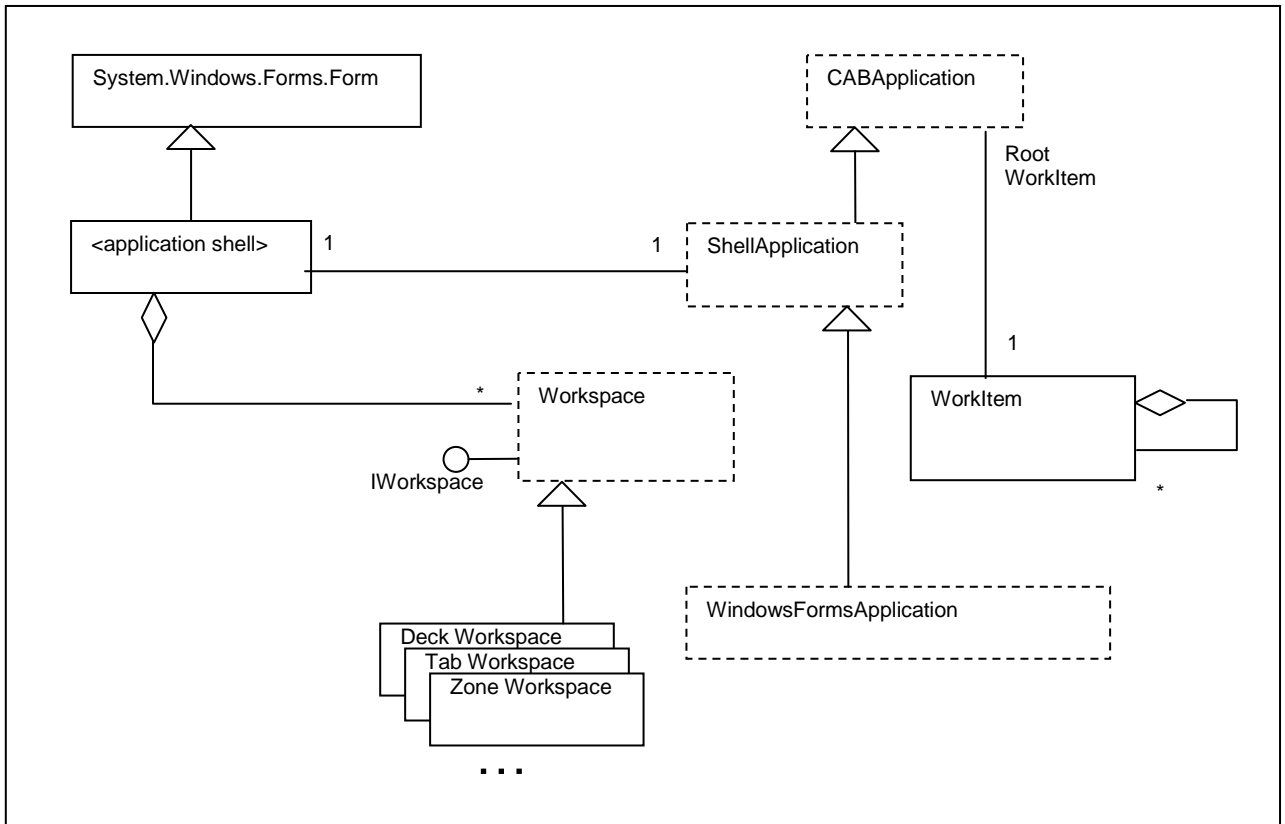


Figure 1: CAB Application Level Class Structure

2.2.2 Classes

This section provides summary descriptions of the functions of the CAB application level classes shown above.

CabApplication

- Defines the lifecycle of a CAB application.
- Supplies the object builder used to create and configure objects throughout the application.
- Sets the initial strategies and policies of the object builder
- Creates and initializes the root WorkItem.
- Initializes some services (required, optional based on configuration, added by child classes)

CabShellApplication

- Holds the reference to the enclosing visual shell of the application.
- Allows the visual shell to inherit from System.Windows.Forms.Form or other visual component appropriate for the UI rendering technology used by the application.

WindowsFormsApplication

- Abstract base class for applications with user interface based on WinForms.
- Sets up forms-based elements of the application

<application shell>

- Actually part of the application, inherits from System.Windows.Forms.Form.
- Provides the top-level Workspace visual component (and possibly others).

Workspace

- Base class for visual container components
- CAB supplies several useful implementations

WorkItem

- The “backbone” of the CAB architecture.
- Holder for facilities needed to perform a use case
- Hierarchical

2.3 CAB WorkItem Level

WorkItems form the backbone of a CAB application. As the application is composed at runtime, a hierarchy of WorkItem instances is created, each supporting a particular use case or sub-use case of the application. Each WorkItem instance serves as a central repository for facilities needed to accomplish its use case and any sub-use cases. These include views (visual presentation and layout components), services, events and commands.

2.3.1 Structure

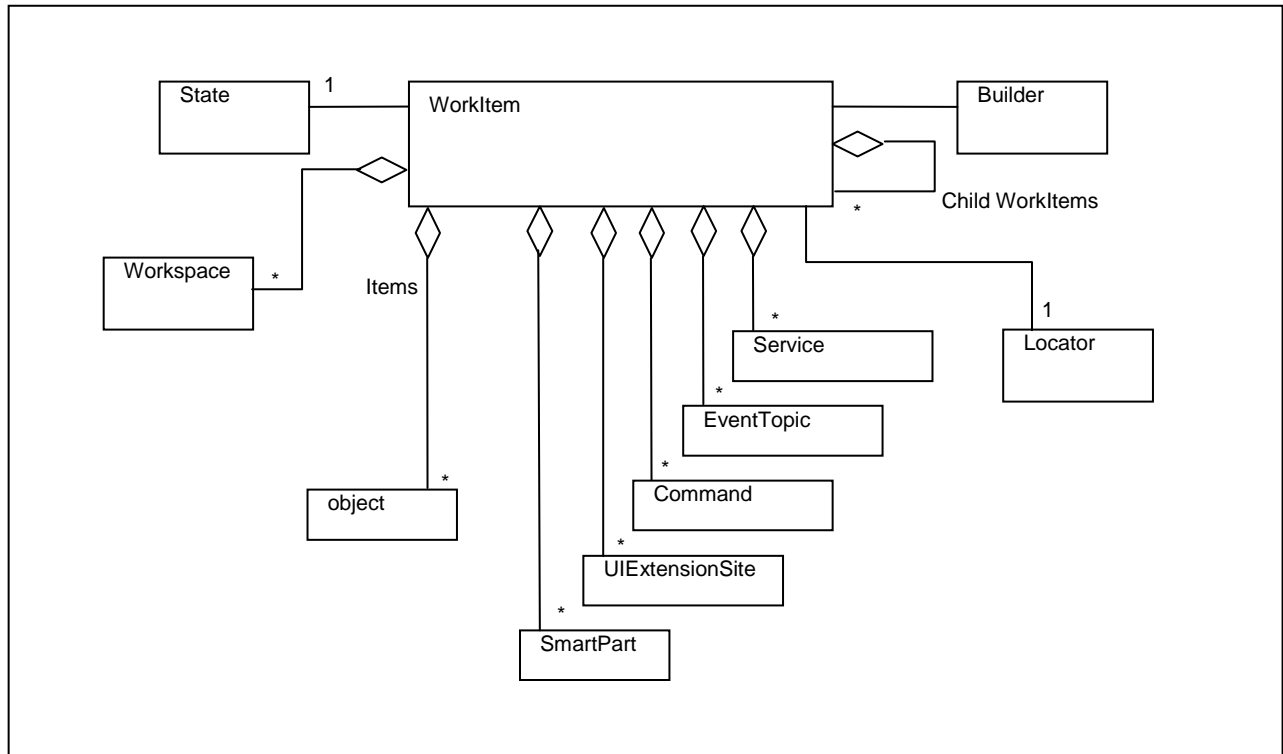


Figure 2: CAB WorkItem Class Structure

2.3.2 Classes

This section provides summary descriptions of the workitem level CAB classes shown above.

WorkItem

- The primary repository for facilities required by use cases.
- Holds objects needed during execution of use cases:
 - State (a bag of state objects local to the WorkItem). The State bag is not used by the IdeaBlade/CAB components and its use by applications is not recommended.
 - Builder (Strategy and policy-based object builder)
 - Services
 - EventTopics
 - Commands
 - UIExtensionSites
 - Items (arbitrary objects)
 - Workspaces (visual containers for controls) – discovered by name and attached at runtime.
 - SmartParts (visual elements hosted by workspaces)
- Hierarchically organized. Facilities of ancestor WorkItems are available to all descendents.

Locator

The WorkItem actually maintains a single collection of objects (“Items”). The specialized collections shown above (SmartParts, Commands, EventTopics, etc.) are actually facades supplied by the Locator class that segment the items collection by object type.

SmartPart

SmartParts are visual components displayed in Workspaces or SmartPartPlaceHolders in views. They are identified by adding the [SmartPart] attribute to their class declaration. Being visual components, SmartParts must inherit from control-suite-specific base classes, so the Locator is unable to distinguish them by a common ancestor class. For this reason the SmartPart attribute *must* be used.

2.4 CAB Module Level

Modules are units of functionality composed into an application from assemblies loaded dynamically at runtime. The modules to be loaded and the order in which they are loaded is determined by the configuration file ProfileCatalog.xml. This simple XML file defines:

- The specific modules to be loaded, possibly based on the current user’s role.
- The assembly containing the module.
- Other modules or groups of modules on which the module depends. Dependency modules are loaded first.

Modules are defined by the presence of a class that inherits from the CAB class ModuleInit. When a module assembly is loaded, each class that inherits from ModuleInit is instantiated and its Load() method invoked. This class can receive control at several points during the application composition process:

- The ModuleInit constructor receives a reference to its parent WorkItem.
- The Load() method may be overridden to perform initializations when the module is about to be loaded into the application. At this point, modules on which this module depends have already been loaded and their facilities are available in the parent WorkItem. The module may create its own child WorkItem or continue to configure the parent WorkItem passed in its constructor.
- The AddServices() method may be overridden to configure module-specific services into the WorkItem.
- By subscribing to the RunStarted event of the WorkItem, the module can again receive control after all modules have been loaded into the application.
- By defining command handlers, the module receives control when specified commands are activated (typically by user action).

2.4.1 Structure

2.4.2 Classes

ModuleInit

The initializer class for a module. May perform extensive initialization. May create a new child WorkItem, supplying WorkItem Controller class.

3 IdeaBlade/CAB

The IdeaBlade/CAB extensions provide commonly-used CAB functionality tailored for integration with DevForce. It includes support for multiple UI rendering technologies and third-party control suites. It supplies many CAB-based components suitable for inclusion in end-user applications.

3.1 Global Definitions

3.1.1 Global Name Definitions

To allow loose coupling between components, most CAB facilities are referenced by name. Components, such as module or WorkItem controllers, that provide facilities add them to a WorkItem where they can be accessed by other components (perhaps in different modules). When an object is added to a WorkItem, it is stored in the Items collection, indexed by its unique name. Based on the object's type (or the presence of a SmartPart attribute) the Locator supplies it through the appropriate collection façade.

Because all objects are stored in a single collection, *all object names must be unique*. To help enforce uniqueness and avoid errors due to misspelling (and to provide Intellisense support), the IdeaBlade/CAB extensions define static classes containing string-valued constants that equate to the names used by its components. Applications extend these classes (within their own namespaces) to define additional names.

The constant classes defined by the IdeaBlade/CAB are shown below. The actual defined names are contained in Appendix I. The suggested naming guidelines are important to ensure that all object names remain unique.

- CommandNames Always end with "Command"
- EntityManagerNames Always end with "EntityManager"
- EventTopicNames Always end with "Event"
- ResourceNames Always end in "Icon", "Image", "Color", "Text" or other type name
 - Logo Images
 - Condition Images Represent conditions such as "Information", "Warning", etc.
 - Operation Images Represent operations such as "Add", "Save", "Undo", etc.
 - RowState Images Represent the state of a row in a grid, such as "Added", "Deleted", etc.
 - Texts Commonly used button label text strings
- UIExtensionSiteNames Always end with "Site"
- WorkspaceNames Always end with "Workspace" or "SmartPartPlaceholder"

3.2 Views

Creating visual presentations of information is a major part of application development. In IdeaBlade/CAB, visual components, called views, are broken into three categories:

Base Views

These function as base classes that provide functionality for application-specific visual presentations.

Layout Views

A layout view provides the “floor plan” of a part of an application’s user interface. It might divide its screen area into areas and provide facilities for displaying, activating and controlling other views displayed in those areas, but typically does not directly contain controls that display user data.

IdeaBlade/CAB provides several commonly used layout views.

Widget Views

Widget views display actual application-specific data using controls or other views.

IdeaBlade/CAB encourages creating small, single-purpose widget views, then composing them at runtime using layout views.

CAB and the IdeaBlade/CAB extensions assist with the creation of rich user interfaces in several ways:

- CAB itself provides the concept of Workspaces and provides several useful examples. Workspaces represent portions of screen real estate. They hold views that in turn may hold other views in containers called “smart part placeholders”. Workspaces may hold multiple views simultaneously and provide facilities for automatic layout and activation/deactivation.
- By defining base, layout and widget views, the IdeaBlade/CAB encourages separating the layout aspects of user interface design from the information presentation aspects.
- IdeaBlade/CAB, building on the classes generated and supplied by SCSF, encourages the use of the Model-View-Presenter paradigm [*reference WB paper?*]. In this approach, the purely visual elements of an information display component are isolated in a *view* class with little or no functional logic. The logic of responding to user input and adjusting the appearance of the display is encapsulated in a separate *presenter* component, visible only to the view. Finally, the source of displayed data is isolated in the *model* component, often in the form a *BindingSource* connected to a data source such as a DevForce entity object.
- IdeaBlade/CAB encourages separating an application’s views into *widget views* that actually present data and allow user interaction, and *layout views* that prescribe the size and positioning of views (either widget views or other layout views).
- IdeaBlade/CAB encourages building fine-grain views that present a small amount of data (often from a single DevForce entity or entity type) in a few controls, then composing these into more complex displays using layout views.

To assist in developers in creating application-specific views, IdeaBlade/CAB provides classes and interfaces that supply much of the CAB “boilerplate” functionality required of all views in the CAB framework. In addition, they provide for smooth introduction of third-party control suite support.

3.2.1 Common View/Presenter/Context Support

Much of the support functionality for views is not control-suite and implemented is common classes provided by IdeaBlade/CAB as shown and discussed below:

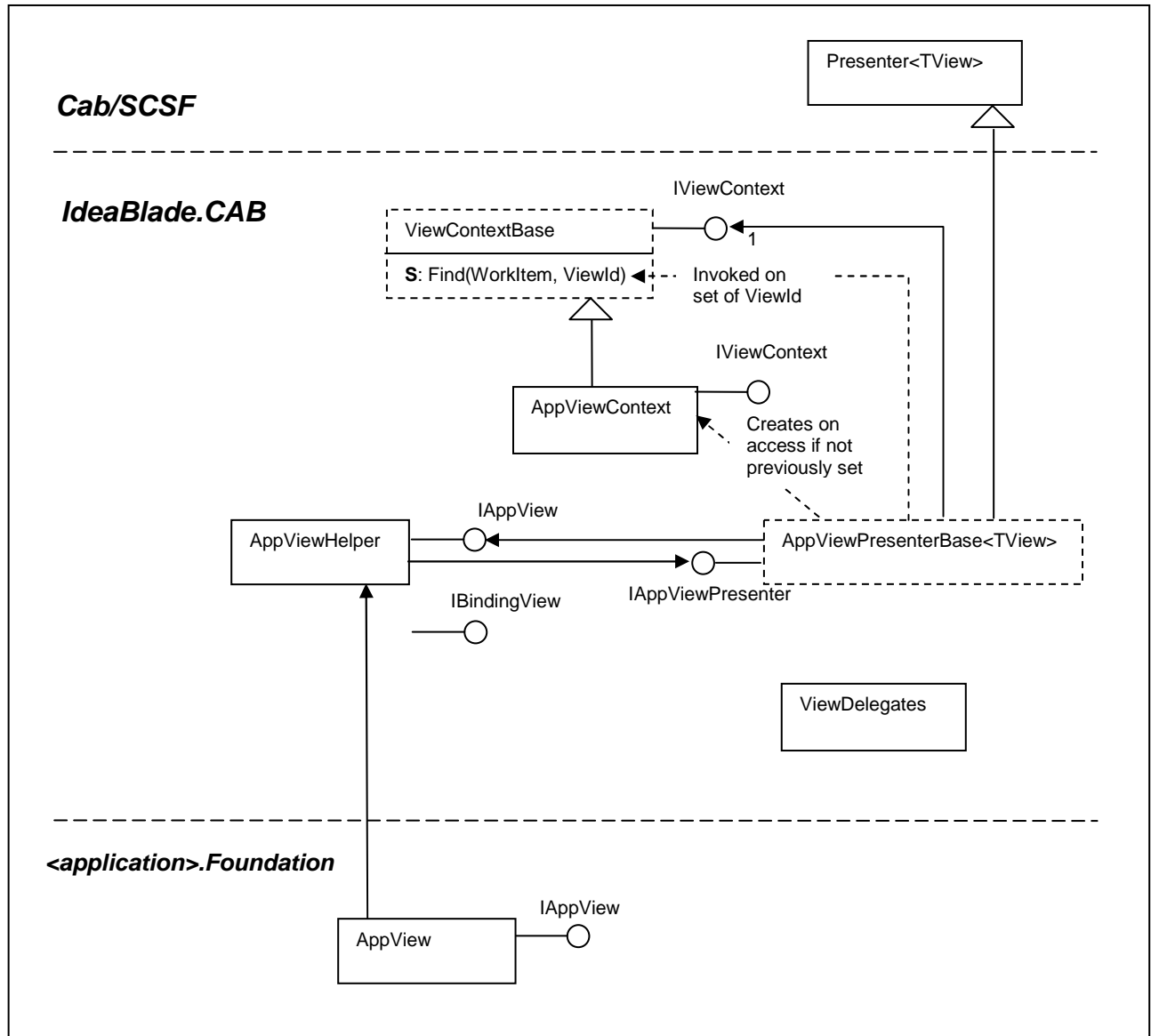


Figure 3: Common View/Presenter/Context Support Class Structure

IAppView

This is the common interface exposed by all IdeaBlade/CAB application views based on the IdeaBlade/CAB extensions. It allows the view to be referenced by the AppViewPresenterBase base class.

IBindingView

This extension of IAppView is exposed by all IdeaBlade/CAB and application views that use data binding.

IAppViewPresenter

Hides the view-specific nature of `AppViewPresenterBase<TView>`. It allows `AppViewHelper` to access base presenter functionality regardless of the type of the view the presenter supports.

AppView

IdeaBlade/CAB defines the `AppView` class to serve as the base class for all application views. `AppView` is actually defined at the application foundation level and determines the “primary” UI rendering technology and control suite for the application.

AppViewHelper

Because all views must ultimately inherit from a visual component (such as `System.Windows.Forms.UserControl`), `AppView` delegates most functions to the `AppViewHelper` class. Through `AppViewHelper`, `AppView` supplies management of the `ViewId`, `SmartPartInfo`, and `ViewContext`. An application may extend `AppViewHelper` to provide additional functionality to application-specific views.

`AppViewHelper` provides the storage for the injected reference to the view’s presenter, but the actual injection must take place in the final, most derived view class where the type of the presenter is known.

ViewContextBase

The IdeaBlade/CAB extensions use a context object to communicate view-specific context information from a page or `WorkItem` controller to a view’s presenter. `ViewContextBase` is an abstract base class that provides basic functionality for this task. Using the injected `WorkItem` and resources, it provides facilities for naming itself (based on its view instance id) and making itself available in a `WorkItem`. A static `Find()` method allows a view’s presenter to acquire the context in which the view should be managed directly from its `WorkItem`.

AppViewContext

The `AppViewContext` class simply provides a default concrete context for a presenter if none has been made available by a page or `workItem` controller.

AppViewPresenterBase<TView>

`AppViewPresenter Base` is an abstract generic class that serves as a base class for all application presenters. Because it is not dependent on the rendering technology or control suite, it is defined in the `IdeaBlade.CAB` assembly.

`AppViewPresenterBase` handles accessing of the `ViewContext` from its `WorkItem`, automatically creating a default `AppViewContext` object if none was supplied by the page or `WorkItem` controller.

`AppViewPresenterBase` also supplies “finish open edits” functionality (delegated to a global `UpdateSource(view)` method) to allow presenters to capture all changes before saving or performing other processing, even if the user failed to exit a control.

ViewDelegates

This class defines delegate types that allow views to receive information from their presenters. An alternative is to add methods to the view's interface, but the delegate technique has several advantages:

- By receiving information through registered delegates, views avoid publicly exposing functionality that should not be invoked by other objects (e.g., a `WorkItem` controller).
- The presenter remains visible only to the view.
- The view's public interface (to its `WorkItem` controller) does not depend on its specific function, so extending a view does not alter its public interface, potentially breaking other implementers of that interface. This allows many views to share a common interface. In fact, with rare exceptions, all views expose either `IAppView` or `IBindingView`.
- Registering of a delegate with the presenter is an opt-in model. This allows multiple views to share a common presenter and respond only to the method calls they understand. It also allows asynchronous extension of a presenter and the views that use the extended method calls.

3.2.2 Control Suite Specific View Support

IdeaBlade/CAB provides for smooth integration of third party control suites into a CAB-based application. An application may use multiple control suites simultaneously, but in practice, most applications use a single control technology. IdeaBlade/CAB provides for defining one such suite (e.g., native dot net WinForms ("DotNet"), DeveloperExpress ("DevEx") or Infragistics ("IG")) as the primary technology to be used by the application. This is accomplished by defining control-suite-specific versions of `AppView` in dedicated assemblies, then declaring the `AppView` class in the skeleton application foundation module supplied by IdeaBlade/CAB and "switching" its parent class using conditional compilation.

Views based on a control suite other than the designated primary suite simply inherit directly from the suite-specific version of `AppView`.

3.2.3 ViewFactoryService

To allow page and `WorkItem` controllers to construct views based on conditions known only at runtime (such as user role), IdeaBlade/CAB provides the `ViewFactoryService` to create and dispense views. Like many services, this is just a simple registry keyed by a unique view name. The view names themselves are defined as global constants (in the `ViewNames` class) to allow compile time checking and help ensure uniqueness. An application or module may define additional views and extend the `ViewNames` class to add additional names. The application or module then registers its views with the `ViewFactoryService` at startup or load time.

A `WorkItem` controller may create its own `ViewFactoryService` (possibly using one of the control suite-specific versions to automatically register the IdeaBlade/CAB views defined for that suite). The `WorkItem` (and any child `WorkItems`) will then be provided with views by that service instance.

These control-suite-specific classes are shown and discussed below.

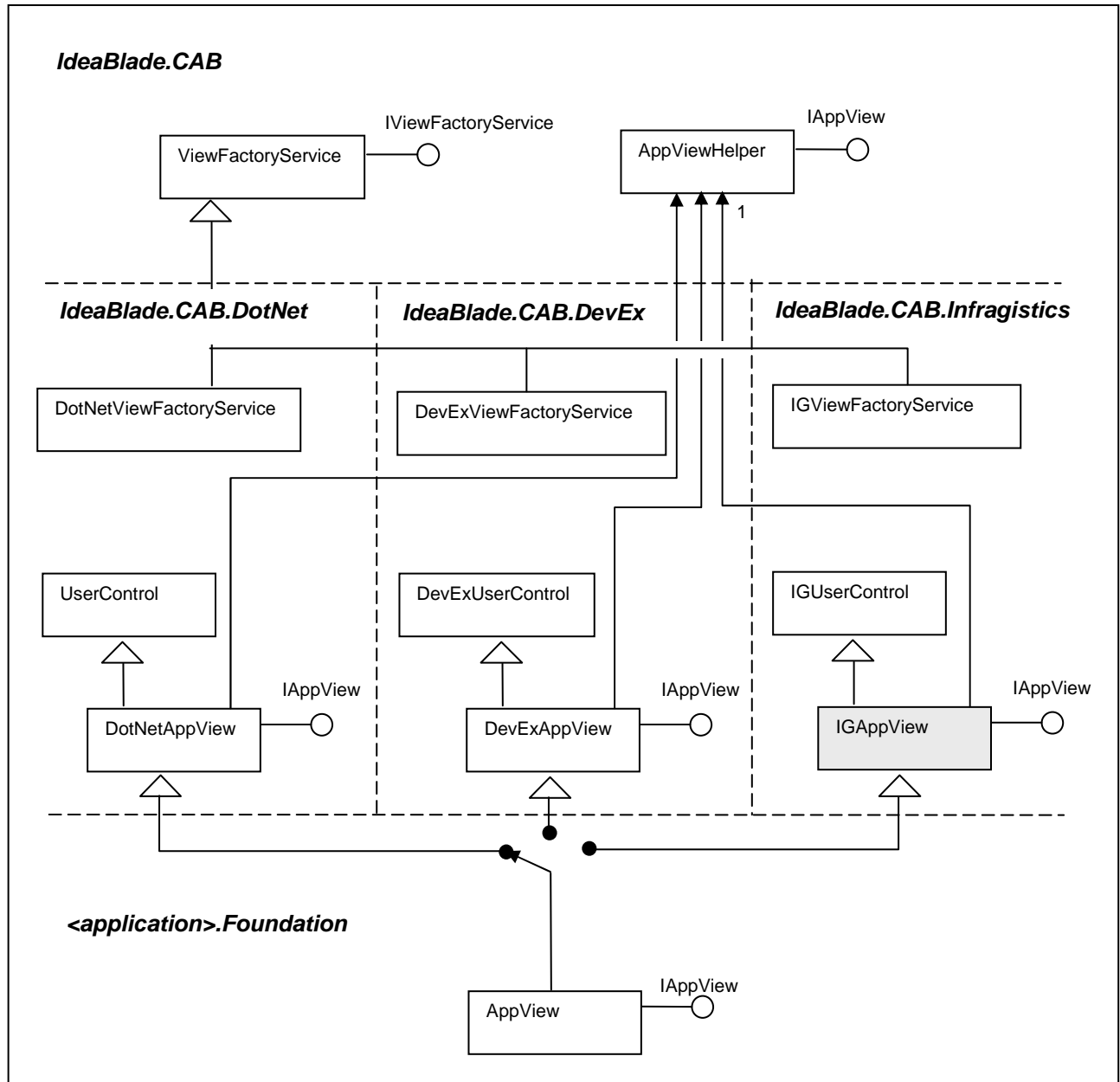


Figure 4: Control suite-specific View Support Class Structure

IViewFactoryService

The `IViewFactoryService` interface is used to register and retrieve views from the view factory service.

ViewFactoryService

This is the concrete implementation of the `IViewFactoryService` interface. It allows registration and unregistration of view types (by name) and handles requests to create instances of views. Method overloads provide for both generic type and method parameters

passed to the created view's constructor as well as the ability to automatically add the created view to a WorkItem.

DotNetViewFactoryService

DevExViewFactoryService

IGViewFactoryService

These classes are derived from ViewFactoryService. Their constructors register the views defined by IdeaBlade/CAB using a particular control suite.

3.2.4 Control Views

In most applications, many views are constructed as a collection of loose controls bound to fields of one or more data items. IdeaBlade's DevForce product includes a ControlBindingManager component that automatically populates a design surface with controls bound to properties of a class or interface. A view may also be constructed manually using the Windows Forms designer and the controls programmatically bound at runtime.

The IdeaBlade/CAB extensions provide classes that perform much of the "grunt work" of managing loose control views across multiple control suites. As with the basic AppView class, a particular rendering technology and control suite is designated as "primary". Views that inherit from the ControlView class (in the application) use the primary technologies. Views that need a different technology or control suite must inherit from a suite-specific ControlView implementation.

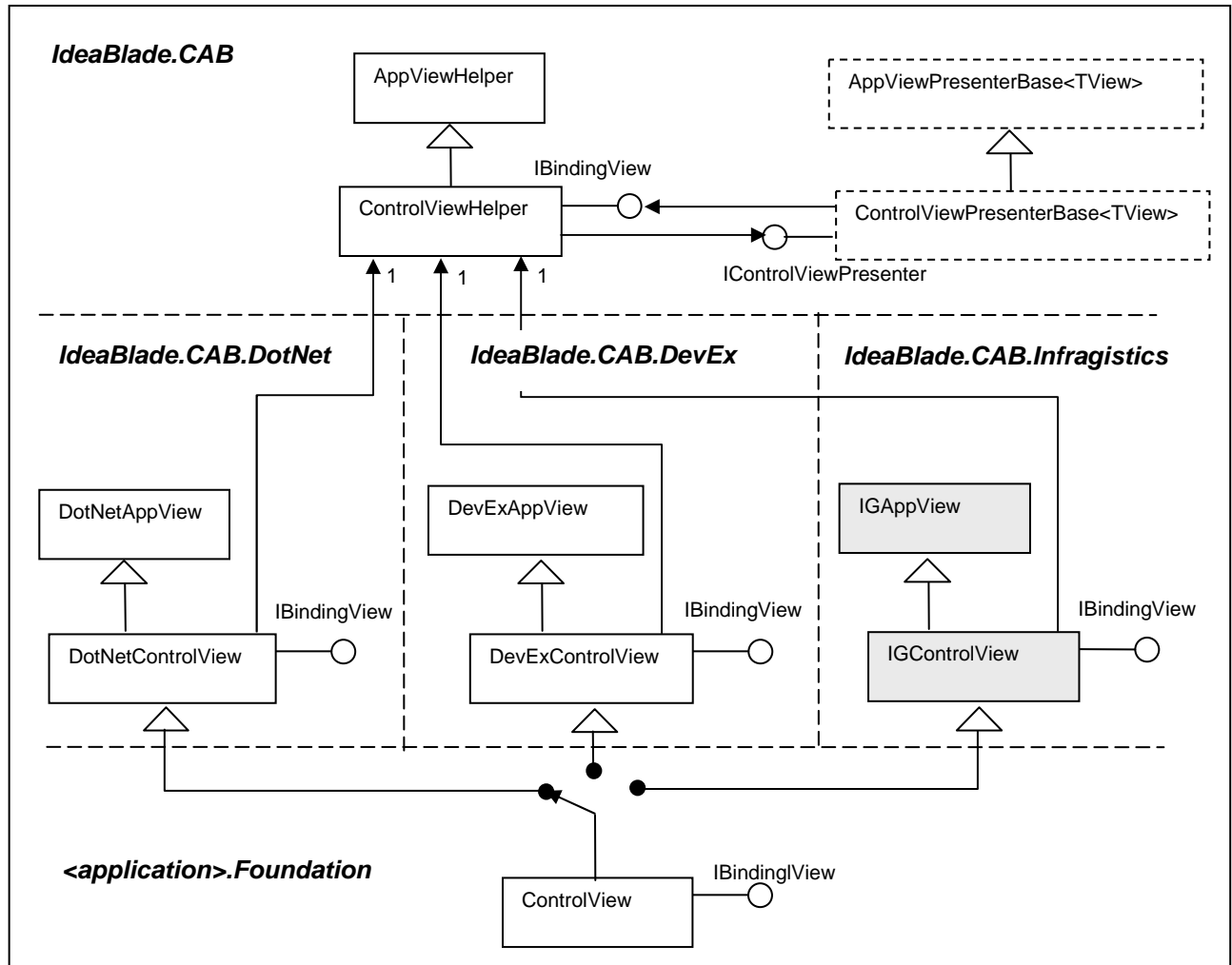


Figure 5: Control View Support Class Structure

ControlViewHelper

In the same manner as AppViewHelper, the ControlViewHelper class provides common functionality for control views independent of their rendering technology and control suite. It provides the implementation of IBindingView and manages the view’s relationship with its presenter.

ControlViewPresenterBase<TView>

The ControlViewPresenterBase class provides common presenter functionality for a control views, regardless of their rendering technology or control suite. If the control view was constructed using a DevForce ControlBindingManager, that binding manager is used. Otherwise, a new binding manager is provided and the controls are bound programmatically via Bind() methods. The presenter automatically delays execution of Bind() calls until the view’s binding source has been set. While implemented, this “automatic” binding manager creation and binding of controls is not recommended.

IControlViewPresenter

Hides the view-specific nature of `ControlViewPresenterBase<TView>`. It allows `AppViewHelper` to access base presenter functionality regardless of the type of the view the presenter supports.

DotNetControlView
DevExControlView
IGControlView

These classes serve as base classes for implementations of `ControlViews` using specific control suites. They are quite small and delegate most of their functions to the common `ControlViewHelper` class. Concrete application-specific control views inherit from one of these (possible via the `ControlView` “switch” class) and add the controls necessary for a specific entity.

3.2.5 Grid Views

Many applications rely heavily on grids to display information in tabular form. Most control suites provide bindable data grid controls to facilitate table presentation. The `IdeaBlade/CAB` extensions provide *GridViews* that encapsulate data grid controls from the supported control suites, often incorporating an additional tool strip for navigation and control of the grid’s appearance and a binding to a specific class or interface. Because `GridView` implementations are often vendor-dependent, these must always be reference in a control-suite-specific version. No “primary” default version is provided.

Binding and configuring a grid view manually using the Windows Forms designer can be a tedious and time consuming process. Moreover, applications often display the same entity data in grid form at multiple places in the user interface. With designer-based manual configuration, each of these must be configured individually. To improve UI consistency and developer productivity, `IdeaBlade/CAB` provides technology, based on the Builder pattern [*reference GOF?*] for populating, binding and styling a grid control at runtime.

Rather than statically defining the columns and styling of a data grid using the WinForms designer, a developer creates a `GridBuilder` class for a specific class or interface (often a `DevForce` entity type) and makes it available via the `GridBuilderService` configured into the `WorkItem` structure. A view developer who needs to display instances of the entity as a data grid merely includes an empty “basic grid” control (for the desired control suite) in the view. Typically, the view’s presenter retrieves the grid builder from the `WorkItem` based on the view’s bound type. Alternatively, the page or `WorkItem` controller that instantiates the view retrieves the appropriate `GridBuilder` from its `WorkItem` and supplies it to the view’s presenter in a `ViewContext` object. In either case, the presenter invokes the `GridBuilder` to populate and style the grid and its columns.

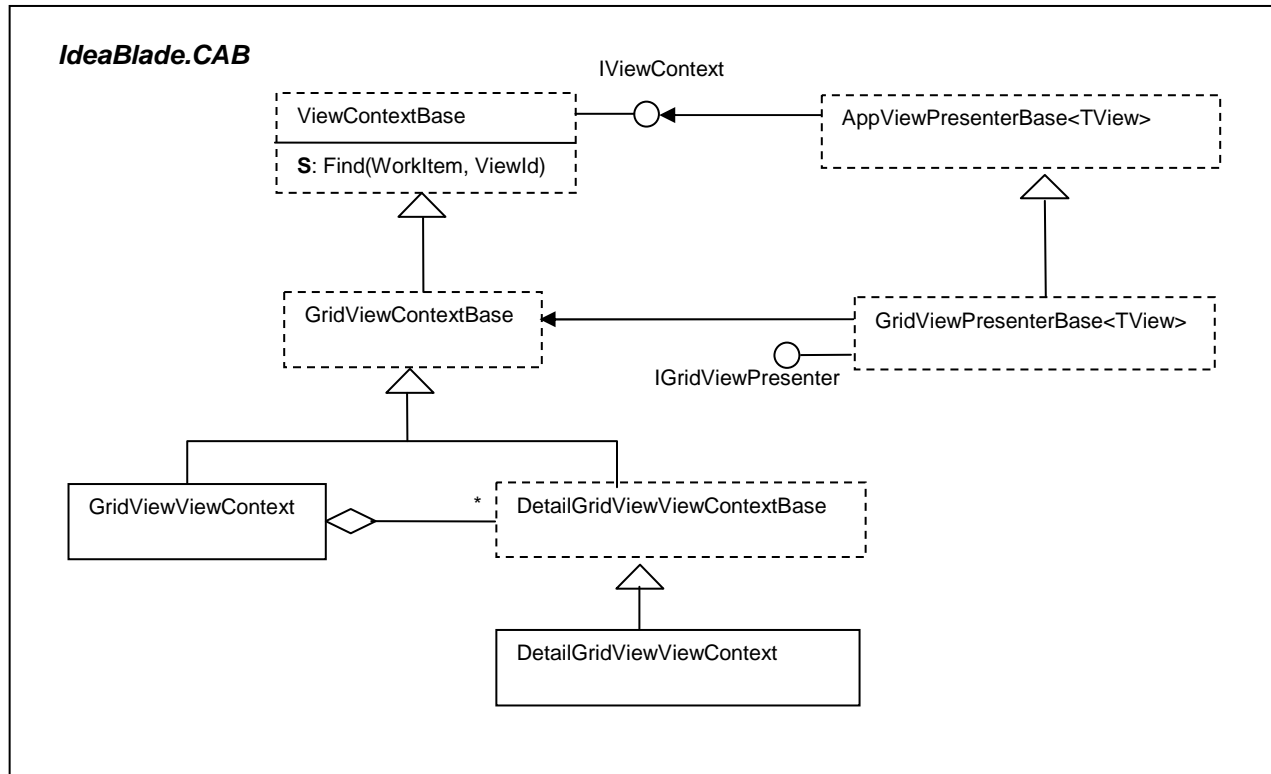


Figure 6: Common Grid View Support Class Structure

GridViewContextBase

The GridViewContextBase class provides context information common to all GridView implementations:

- The name of command to be invoked on row double click
- Flags that control the appearance and behavior of the grid:
 - Grid tool strip visible and/or permanently disabled
 - Read only flag
 - Allow user to add rows flag
- Contexts for detail grids (for grid implementations that support drill-down detail grids)
- Optionally, the GridBuilder to be used to populate and style the grid.

In addition the GridViewContext supplies change notification events monitored GridView presenters. The context raises these when certain properties are changed (typically by the page or WorkItem controller) to dynamically alter the appearance or behavior of the GridView.

GridViewContext

The GridViewContext class is a concrete implementation of the GridViewContextBase class. It provides a static factory method that creates an instance and adds it to a specified WorkItem.

GridViewPresenterBase

The GridViewPresenterBase class provides presenter functionality common to all grid view implementations regardless of the control suite supplying the underlying controls. It processes initialization parameters from view context and sets up handlers for change events on dynamic context data.

The presenter retrieves the appropriate GridBuilder (either directly from the GridBuilderService based on the view's bound type or from the context supplied by the page or WorkItem controller) and invokes it to populate and style the columns of the grid control, creating a binding manager if necessary.

An alternative is to statically populate the grid using a DevForce GridViewBindingManager and the WinForms designer, but this is not recommended.

In either case, the presenter supplies the BindingSource and sets up the connections between the navigation tool strip and the BindingManager.

3.2.6 Control Suite-Specific Grid Views

The implementation of a grid views depends on the capabilities of the underlying data grid control supplied by a particular control suite. Accordingly, IdeaBlade/CAB supplies multiple concrete GridView implementations in separate assemblies. Application projects reference one or more of these assemblies to access grid views for different control suites.

All of the concrete GridView implementations have the same structure. In the following discussion, the string "<C>" represents one of the supported control suites:

- DotNet Controls provided by Microsoft .NET version 2.0.
- DevEx Controls provided by DeveloperExpress (XtraGridView), (currently version 7.1)
- IG Controls provided by Infragistics (UltraGridView).

As other control suites and/or UI rendering technologies (e.g., WPF) are supported by IdeaBlade/CAB, additional assemblies and GridView implementations will be added.

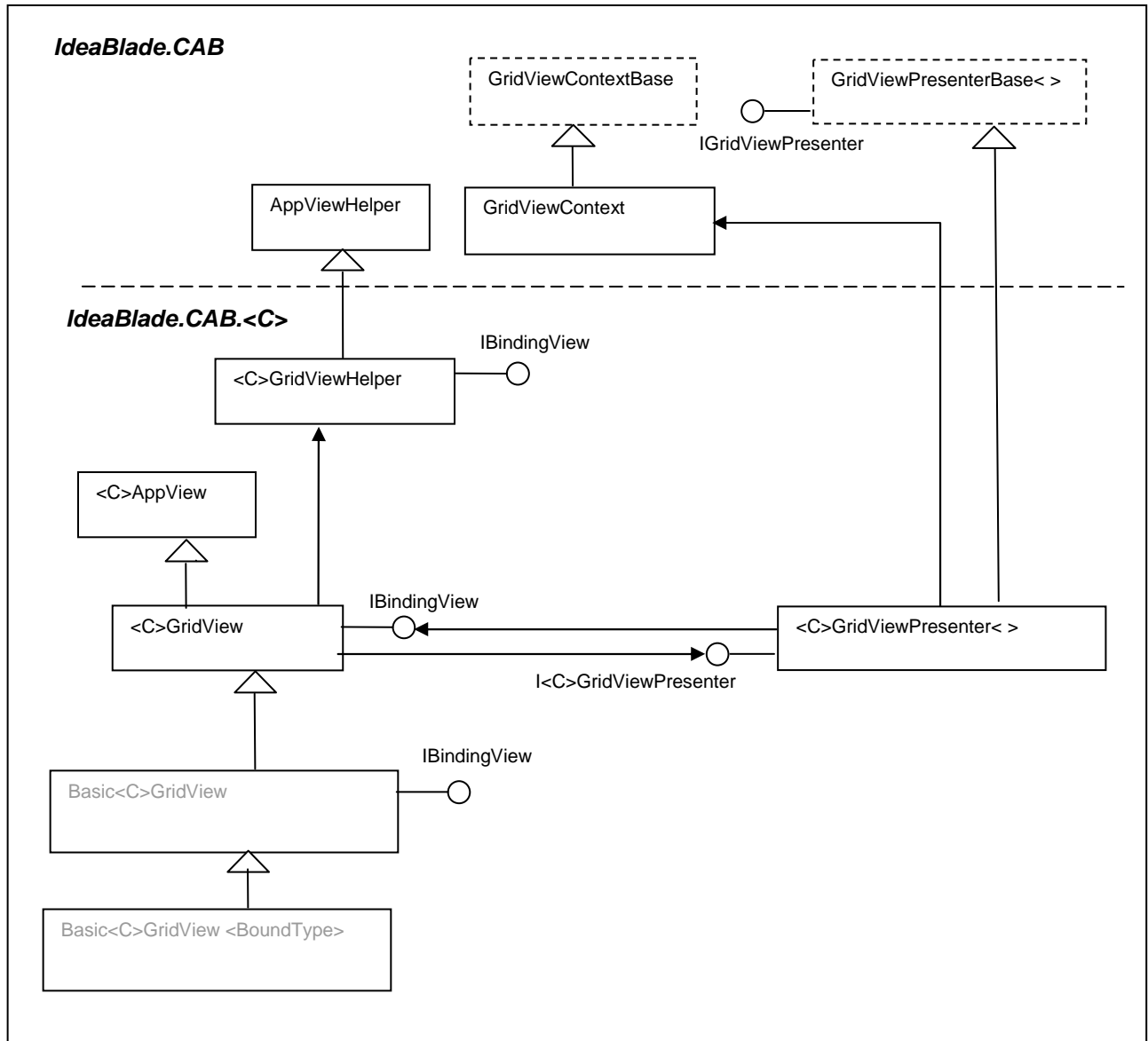


Figure 7: Control Suite-specific Grid View Support Class Structure

<C>GridView

The <C>GridView class brings together the visual components of a grid view for a specific control suite, typically a data grid control and a navigation tool strip. It delegates most of its functionality to the <C>GridViewHelper class.

<C>GridViewHelper

This helper class (currently control-suite-specific) provides the functionality for control-suite-specific concrete grid views. It manages the injected presenter linkage and the BindingManager used by the data grid. It also establishes the linkage between the navigation tool strip and the binding manager.

<C>GridViewPresenter<TView>

The <C>GridViewPresenter class provides the concrete implementation of grid view presenter functionality for a specific control suite's data grid implementation. All implementations provide column sorting capabilities, and this class handles any resulting synchronization issues.

The DeveloperExpress and Infragistics implementations of the grid view control support drill-down to detail views. For these implementations, the presenter responds to the presence of detail grid contexts in the supplied GridViewContext object and creates appropriate detail grid views as necessary.

I<C>GridViewPresenter

The type of GridViewPresenter<TView> objects is unique to the concrete grid view it serves. In order for <C>GridViewPresenters to be accessible to the base <C>GridView and GridViewHelper classes, they expose the common I<C>GridViewPresenter interface.

Basic<C>GridView

The Basic<C>GridViewClass supplies a concrete control-suite-specific grid view implementation whose constructor accepts the bound type at runtime. The bound type is typically a DevForce entity type, but can be any type implementing IList. The primary purpose of this class is to provide for injection of the presenter, in this case of type <C>GridViewPresenter<IListView>. Other application concrete implementations of <C>GridView might inject different types of presenters.

Basic<C>GridView.Generic

This generic version accepts the bound type at compile time and merely passes it through to the base Basic<C>GridView class constructor. It allows the object builder to handle both the instantiation and adding to the WorkItem. Unfortunately, this class cannot be placed on a design surface because it causes the Windows Forms designer to blow up.

3.2.7 Task Navigation Views

Many applications use a central navigation paradigm to guide the user from task to task. IdeaBlade/CAB provides support for navigation through *navigation views*. The visual components of the navigation views are dependent on the control suite in use and the particular navigation technology (e.g., Outlook Bar). Much of the control and task linking logic, however is technology independent.

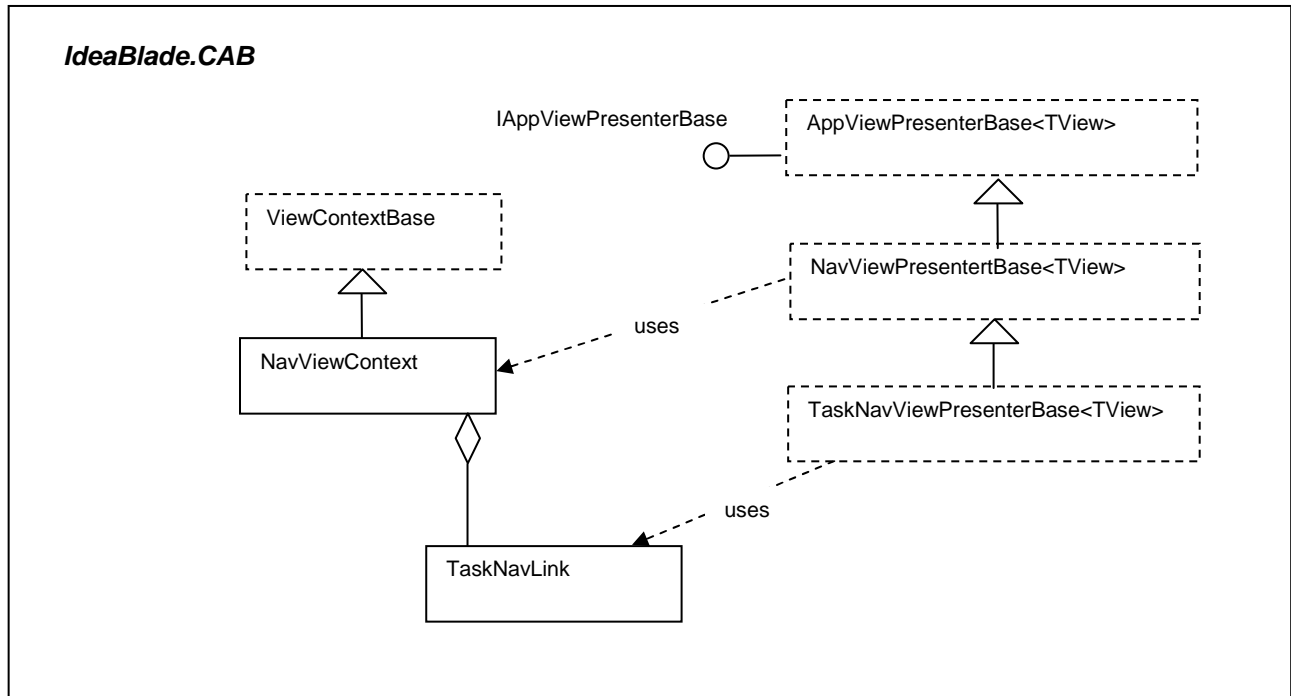


Figure 8: Task Navigation View General Support Class Structure

NavViewPresenterBase<TView>

This abstract class provides the base interface for navigation views. It accepts a NavViewContext object but does not make use of its list of TaskNavLinks.

TaskNavViewPresenterBase<TView>

TaskNavViewPresenterBase specializes NavViewPresenterBase to make use of the list of TaskNavLink objects maintained by NavViewContext. It provides generic task navigation functionality used by control-suite-specific presenters.

NavViewContext

NavViewContext supplies specific context information for a navigation view. This includes:

- NavViewSmartPartInfo Information used by the navigation view’s container to control the visibility and position of the view.
- A list of TaskNavLink items representing the tasks available for selection by the user.

The constructor accepts a WorkItem to which the new NavViewContext will be added with an identifier derived from the id of the TaskNavView it will parameterize. If no view id is supplied, the default view id is obtained from the NavBarService.

TaskNavLink

A TaskNavLink represents a task entry in a navigation view. It includes the name of the task and an optional image to be displayed to the user, plus an event handler to be invoked when the task is selected by the user.

3.2.8 Control Suite Specific Navigation Views

Much of the implementation of task navigation is dependent on the particular control suite and controls used. IdeaBlade/CAB supplies implementations for popular control suites in separate assemblies.

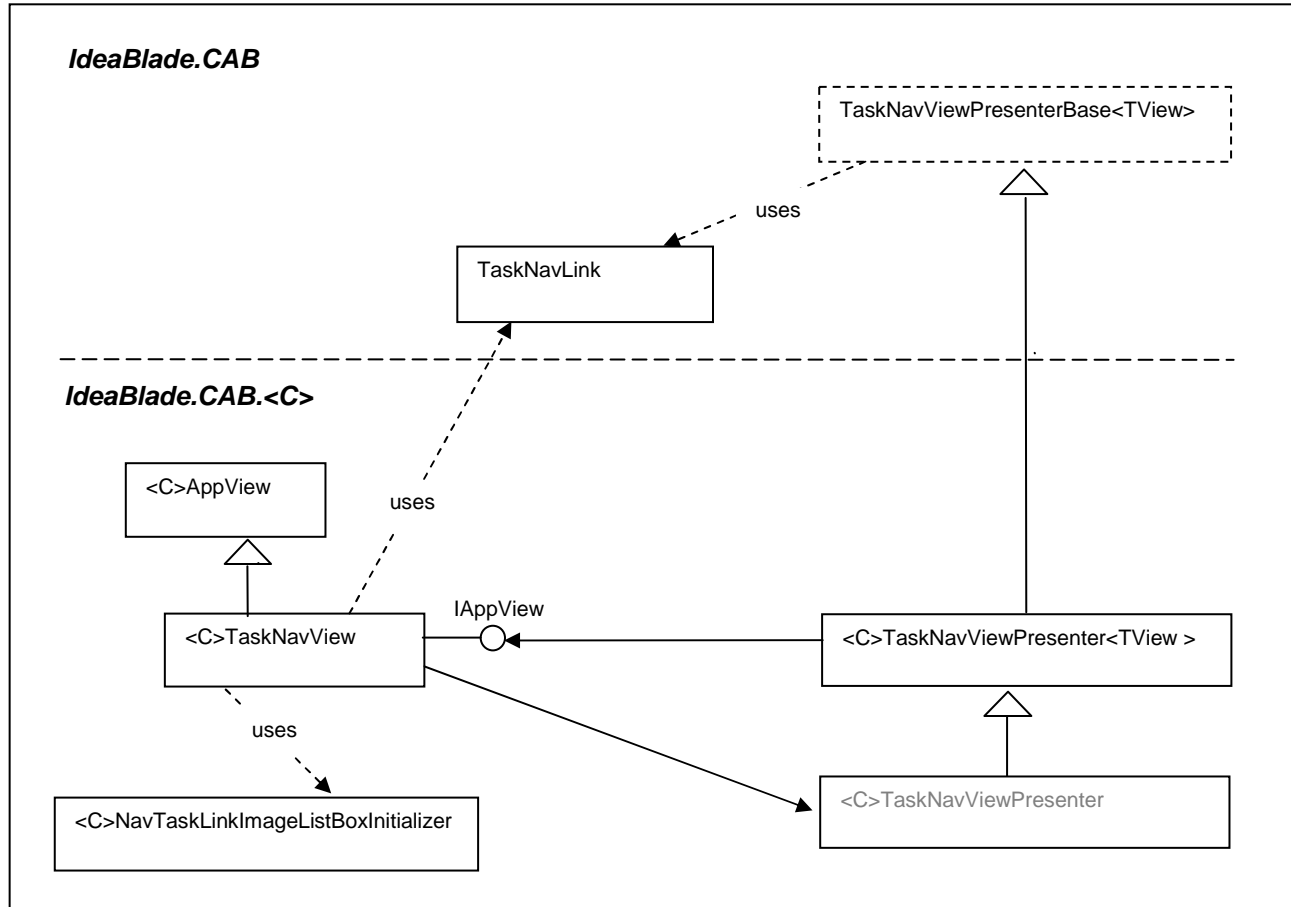


Figure 9: Task Navigation View Control Suite Specific Class Structure

<C>TaskNavView

IdeaBlade/CAB provides concrete implementations of TaskNavView for supported control suites. Like all good views, these delegate most of their functionality to their presenters.

<C>TaskNavViewPresenter<TView>

These are control suite-specific implementations of presenter logic for task navigation views. They contain very little functionality.

<C>TaskNavViewPresenter

This empty class merely provides a non-generic alias for <C>TaskNavViewPresenter<IAppView>.

ImageListBox

The native DotNet control suite does not contain a list box capable of displaying images along with text. This class provides that functionality as part of IdeaBlade/CAB.

NavTaskLink<C>ImageListBoxInitializer

The logic of initializing the elements of a task navigation view is somewhat complex and view-specific (it interacts directly with the suite-specific controls). It is placed in a helper class to keep the view as simple as possible and allow it to be automatically tested. This class provides a Initialize() method that provides all of the initialization functionality.

3.2.9 Default Navigation Implementation Selection

Like the AppView and ControlView, the application foundation skeleton declares classes that select the default control suite for task navigation views by inheriting from the appropriate control-suite specific class.

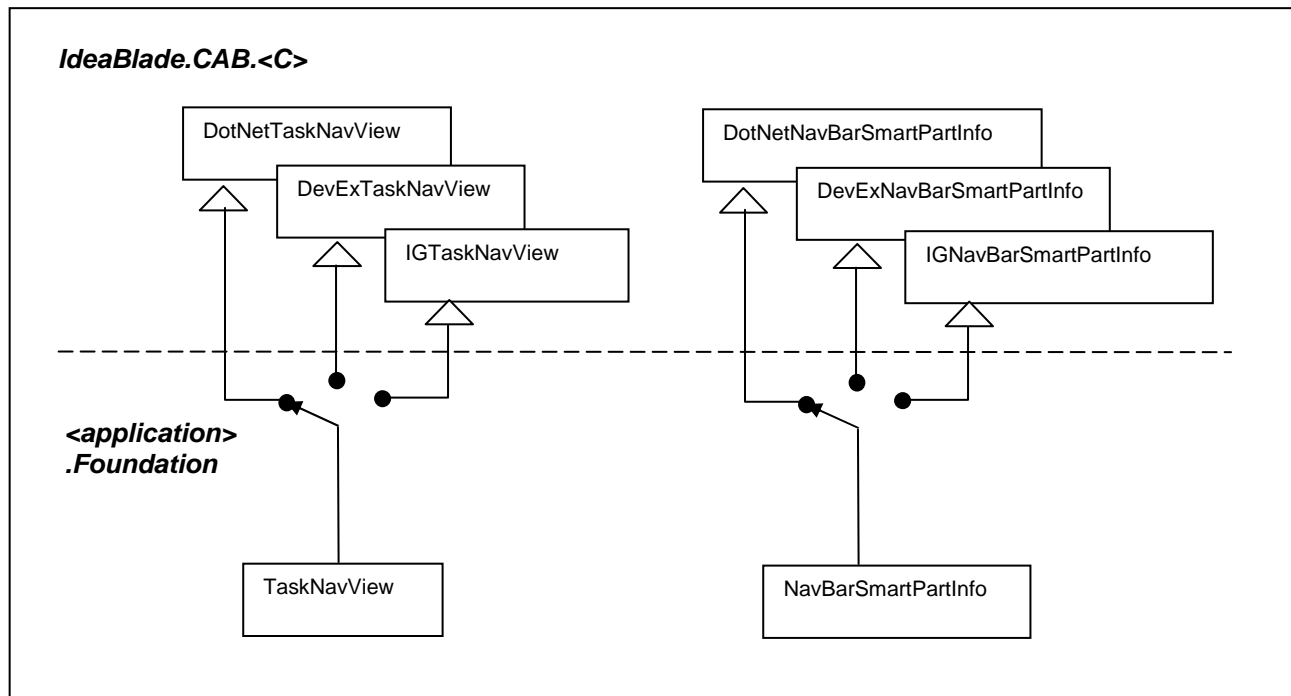


Figure 10: Selection of Primary Control suite-specific TaskNavigation Implementation

3.2.10 NavBarService

In keeping with the CAB architecture, task navigation is implemented as a module composed into the application at runtime. To decouple an application's tasks from its navigation implementation, IdeaBlade/CAB provides the NavBarService (typically configured into the root WorkItem by the module supplying the navigation structure).

Several factors influence the construction of a navigation view:

- The loading order of modules is governed by their dependencies and the current user context and cannot be changed.

- The desired order of display of navigation items might not correspond to the module load order.
- The order in which NavBars are displayed cannot be changed after they have been added to the navigation view.

To circumvent these issues, the NavBarService employs a queue to delay adding NavBars until all modules have been loaded:

1. When each module is loaded its module controller creates a NavController (with an appropriate SmartPartInfo to control how its view is displayed) and places it in the module's WorkItem. It adds TaskNavLinks to the context for the tasks supported by the module, setting their event handlers to invoke appropriate methods within the module.
2. The module controller instantiates a task navigation view and adds it to the WorkItem, then places it on the queue maintained by the NavBarService of task navigation views to be displayed once all modules are loaded.
3. The module supplying the navigation structure subscribes to the RunStarted event of the root WorkItem. When this event fires, all modules have been loaded by and the NavBarService is directed to display all queued navigation bars in the "Shell Navigation" workspace. This queuing mechanism allows the all navigation bars to be available before they are added to the shell navigation workspace so that they can be sorted (based on the display order contained in their SmartPartInfo objects).

3.2.11 General LayoutViews

IdeaBlade/CAB provides several commonly used layout views. With the exception of the SearchAndResultsPage view, these are currently implemented using native DotNet components. Ultimately, control suite-specific versions will be created to incorporate additional formatting capabilities.

All of the general layout views are implemented using the standard IdeaBlade/CAB class structure:

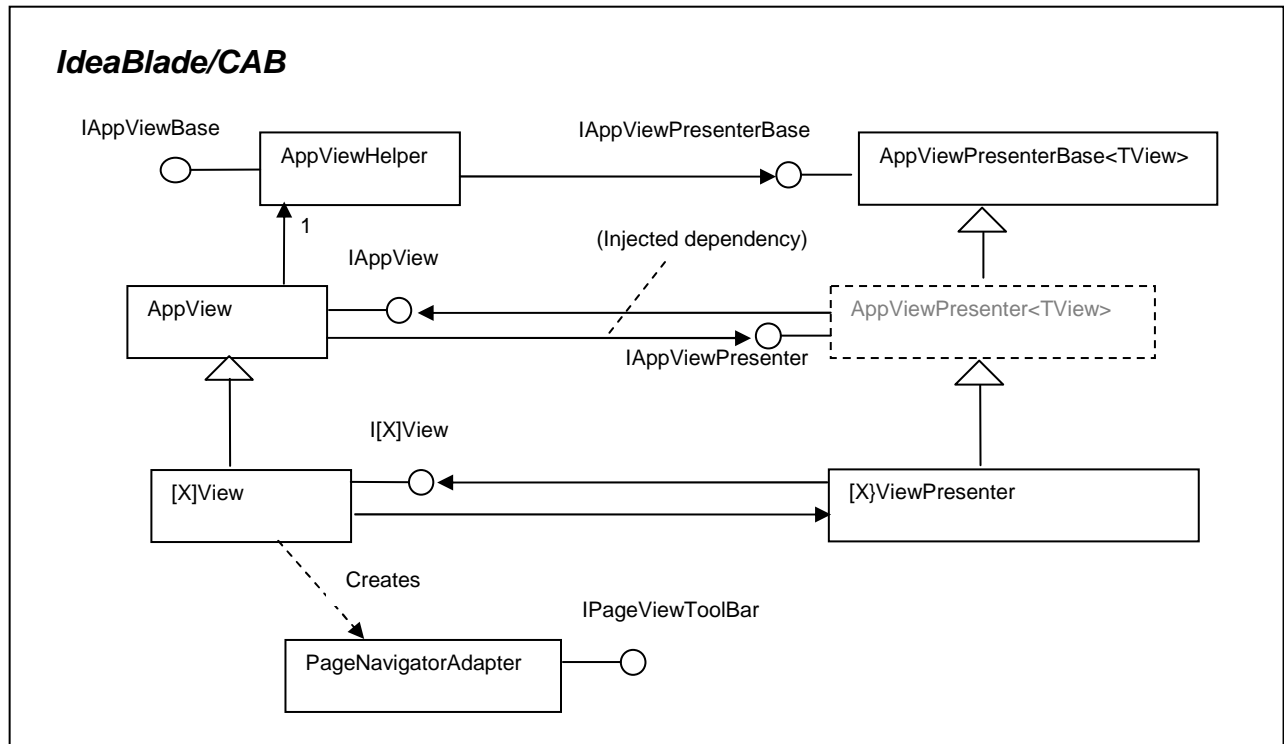
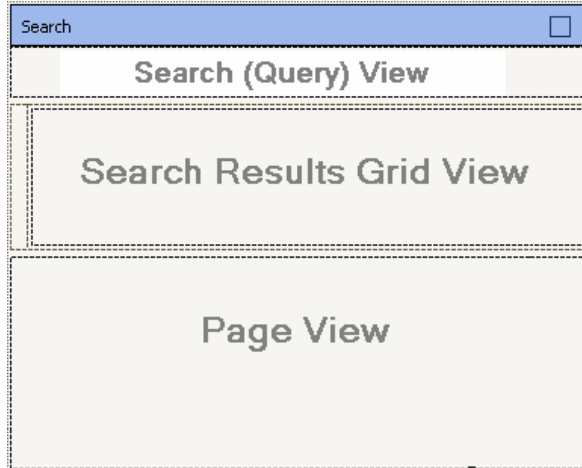


Figure 11: General Layout View Class Structure

The available layout views include:

3.2.11.1 SearchPageView

Available for DevEx and DotNet

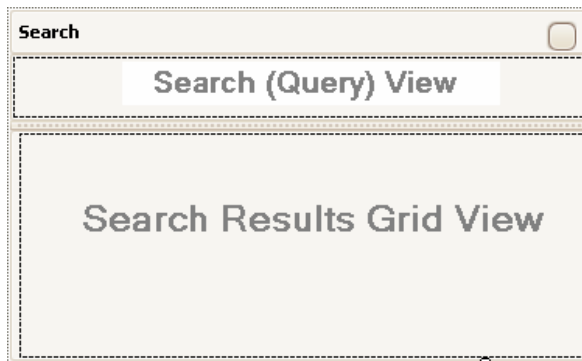


A layout view in which the upper part is a collapsible search area (the Search View + the Search Results Grid View) and the lower part is a “Page” showing the currently selected item in the search results grid.

A SummaryDetail view is the typical occupant of the Page View area.

3.2.11.2 SearchAndResultsPage

Available for DevEx and DotNet

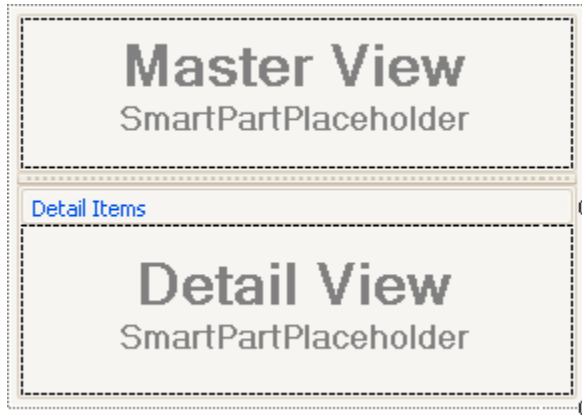


A layout view with a collapsible Search View. The Search Results are always visible. The user can move about in the grid and edit. User should be able to save.

When the view is displayed with a DevEx grid and supplied with detail grid specification, the user can “drill down” into the detail grid(s).

3.2.11.3 MasterDetailView

Available for DevEx and DotNet



A MasterDetailView consists of a horizontal splitter container with a Master SmartPartHolderPlaceholder on top and a Detail SmartPartPlaceholder on the bottom. There is nothing particularly “master-detail” oriented about this view except the names.

3.2.11.4 PageView

Available for DevEx and DotNet



A PageView is a UI container for a semi-autonomous large unit of work that can navigate among a collection of objects that are shown or edited.

The SplitContainer upper panel holds the page body, the main "unit of work" for the Page controller. It displays a single object (typically an entity), the current object in the page's BindingSource.

The navigator (which can be hidden) helps the user move among the displayed objects in the main BindingSource. It also holds buttons for adding, deleting and saving.

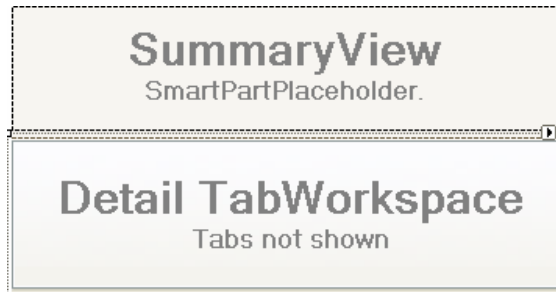
These are wired in an eccentric way that does not take advantage of UIExtensionSite technology yet. It is due for an upgraded approach.

The SplitContainer lower panel holds "alerts" that arise during user's interaction with the page.

At this writing, it displays errant VerificationResults (VRs) for the currently displayed entity. In the not-too-distant future, it will display VRs of other objects related to the current entity as well.

3.2.11.5 SummaryDetailView

Available for DevEx and DotNet

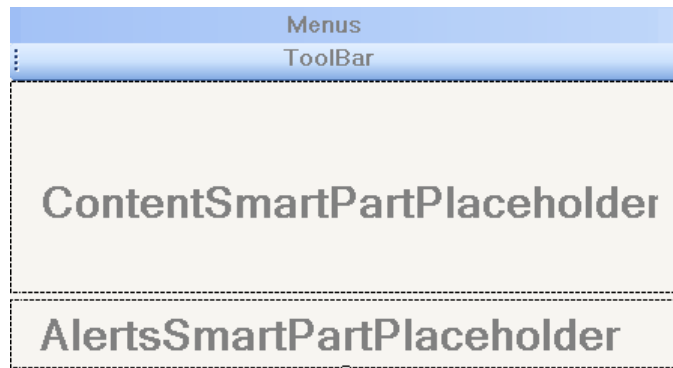


A SummaryDetailView is a view of an item in a collection of objects with the item's summary area above its details area.

The item is the “root object” of the view, perhaps one from a collection of entities such as Employees. The TabWorkspace below presents details of the root object possibly including views of its related objects..

3.2.11.6 EditorView

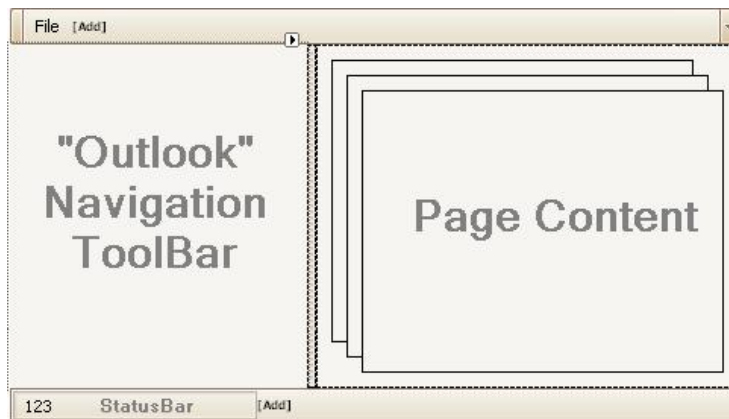
Available for DevEx and DotNet



An EditorView is a UI container for editing a root object in a separate Window. The view consists of a Menu and ToolBar above a horizontal SplitContainer. The SplitContainer upper panel holds the page body, the "unit of work". The SplitContainer lower panel holds "alerts" that arise during user's interaction with the page.

3.2.12 Shell Layout Views

IdeaBlade/CAB provides a common top level shell layout view comprising a navigation pane at the left (the “ShellNavigation” workspace) and a content pane (the “ShellContent” workspace) at the right, together with a main menu and status bar.



The implementation of the shell layout view is dependent on the control suite used for the application. IdeaBlade/CAB provides multiple implementations for the supported control suites. All implementations provide the same functionality.

The presenter for a shell layout view implements the `IShellExtensionService` interface and registers with its `WorkItem` as a service provider. This allows modules to add views to the navigation and content workspaces and interact with the menu and status bar without knowing the control suite used to implement the shell layout view.

Core menu items (like Exit and About) are handled in the shell. They are dispatched on the user interface thread so that they can directly call Form methods without using `Invoke`. The presenters register `UIExtensionSites` with the `WorkItem` to allow modules to extend the user interface.

The shell layout view presenters also register for status update events and use a delegate provided by the view to update the text in the status bar. Modules can fire status update events without worrying about how they will be handled.

3.2.12.1 Developer Express Shell Layout View

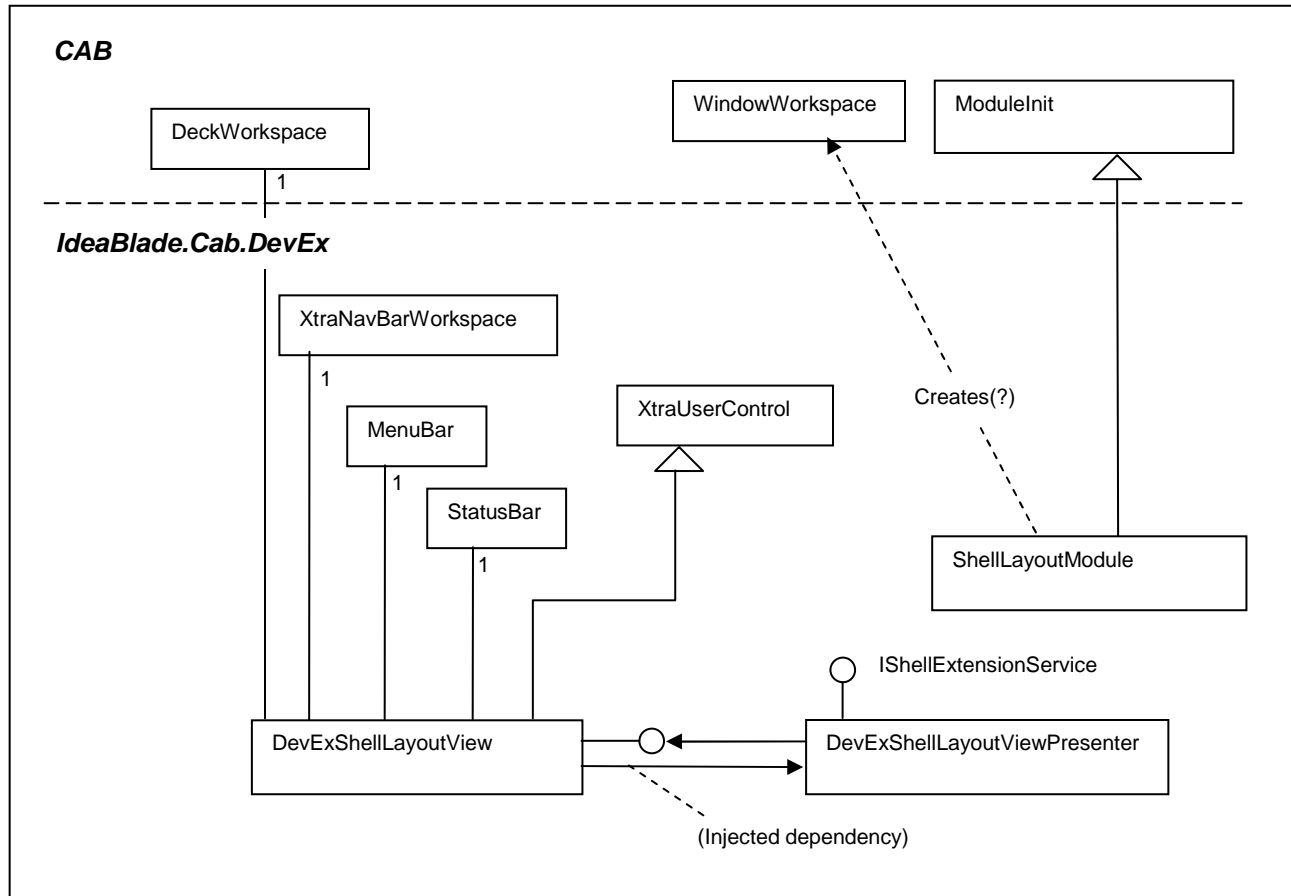


Figure 12: Developer Express Shell Layout View Class Structure

DevExShellLayoutView

The implementation of ShellLayoutView for the DeveloperExpress control suite supplies common UI elements of the application based on DevEx controls:

- Menu and status bars and associated managers (DevX).
- An Xtra splitter control holding the navigation and content panes.
- An XtraNavBarWorkspace for the navigation pane.
- A standard CAB DeckWorkspace for the content pane.

DevExLayoutViewPresenter

The DevExLayoutViewPresenter provides interaction logic for the DevEx shell layout view implementation. It implements and registers IShellExtensionService with its WorkItem, allowing other modules to access the shell layout in a control suite-agnostic way.

3.2.12.2 Dot Net Shell Layout View

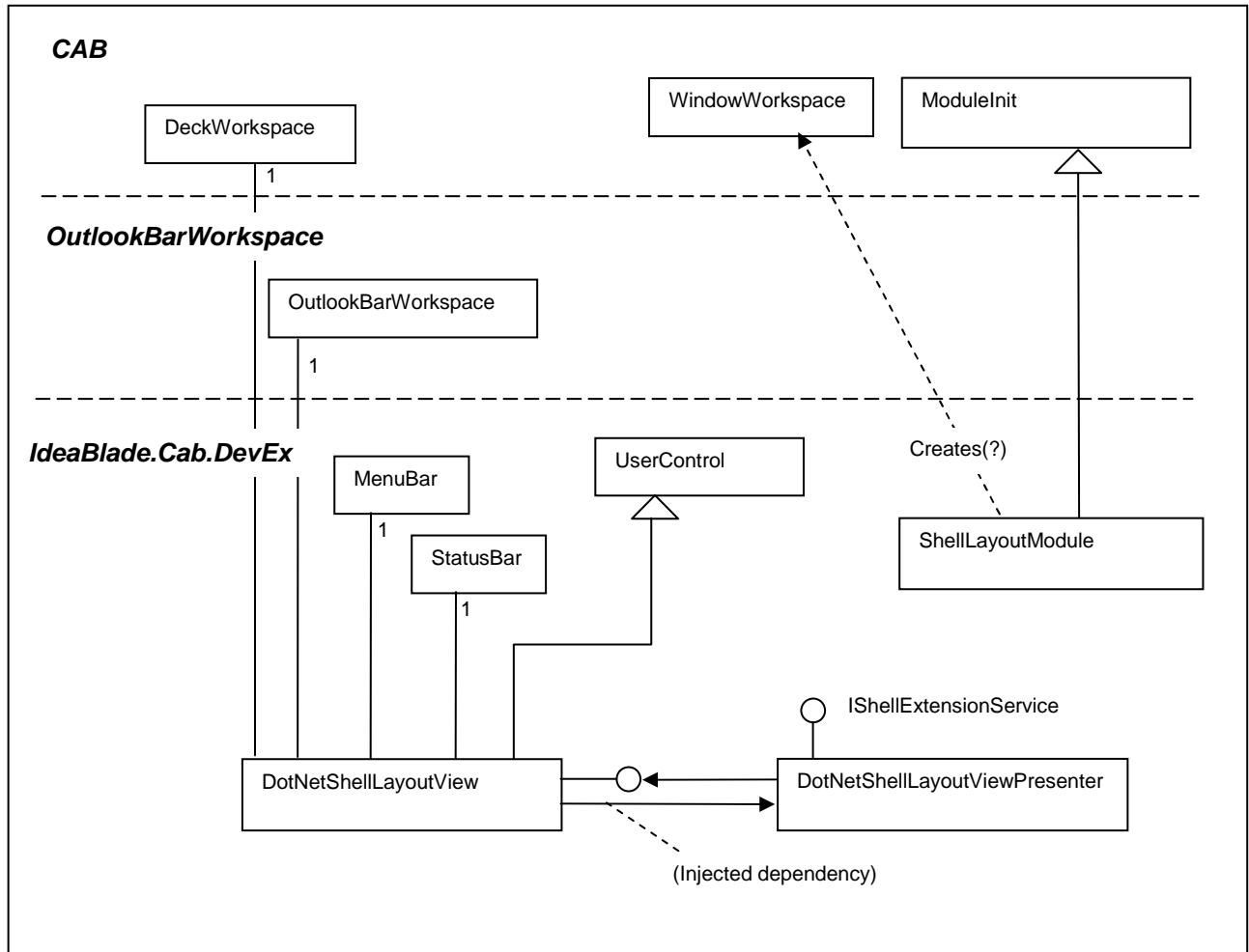
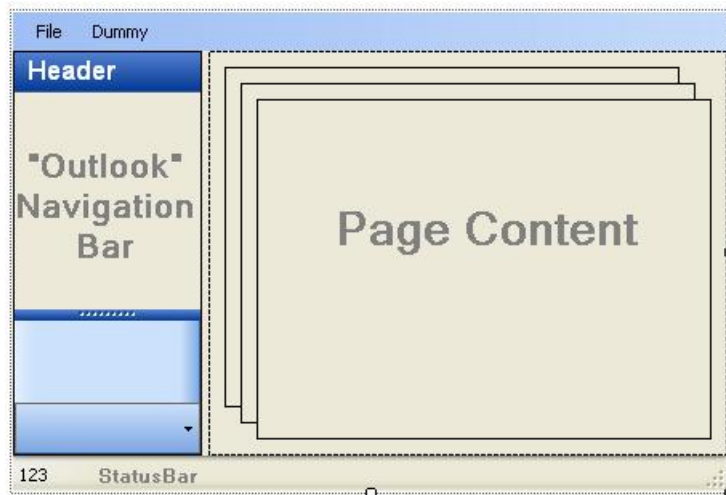


Figure 13: Dot Net Shell Layout View Class Structure

DotNetShellLayoutView



The implementation of ShellLayoutView using native controls supplied by the Dot Net framework. The DotNetLayoutView supplies common UI elements of the application:

Menu and status bars and associated managers (DevX).

- A DotNet splitter control holding the navigation and content panes.
- An OutlookBar workspace for the navigation pane.
- A standard CAB DeckWorkspace for the content pane.

DotNetLayoutViewPresenter

The DotNetLayoutViewPresenter provides interaction logic for the DotNet shell layout view implementation. It implements and registers IShellExtensionService with its WorkItem, allowing other modules to access the shell layout in a control suite-agnostic way.

3.3 Grid Builders

Rather than populating and styling data grids statically using binding managers and the WinForms designer, IdeaBlade/CAB encourages building data grid controls at runtime using GridBuilder classes supplied by a GridBuilder service. Creation and use of a grid builder use the following steps:

- Using facilities supplied by base classes, a developer creates a concrete grid builder class (targeted at a data grid control from a particular control suite) to populate and style a data grid for a bound type.
- During application startup the GridBuilderService is configured into the root WorkItem and populated with grid builder instances, indexed by name (typically the name of their bound type). These are called *prototype* grid builder instances.
- When a grid view has been loaded it informs its presenter (via ViewReady()). The presenter requests a usable GridBuilder instance from the GridBuilderService (passing the bound type and the WorkItem). (Alternatively, the prototype GridBuilder might be supplied in the GridBuilderContext object provided by the page or WorkItem controller.)
- The GridBuilderService retrieves the prototype GridBuilder for the specified bound type and invokes its Create() method to create a copy and marry it with the BindingManager supplied by the presenter. This is called a *working* GridBuilder. It is added to the WorkItem and returned to the GridViewPresenter.
- The presenter invokes the Build() method of the GridBuilder to actually populate and style the data grid control.

3.3.1 Common Grid Builder Classes

IdeaBlade/CAB supplies base classes that facilitate creating grid builders for specific data binding sources. Much of the logic is common to all control suites.

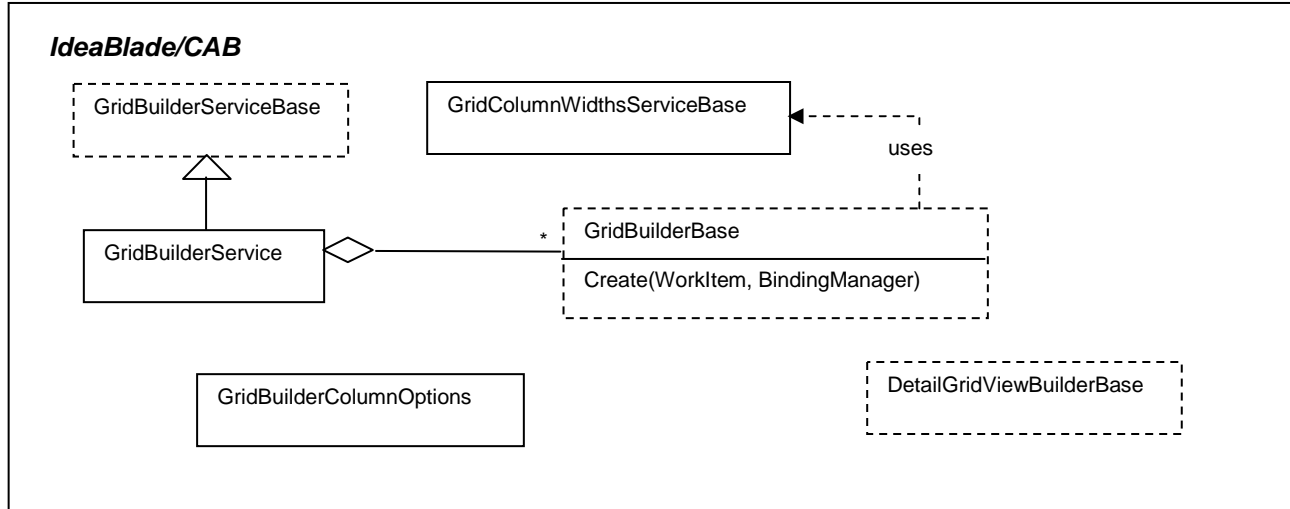


Figure 14: Common Grid Builder Class Structure

GridBuilderBase

The GridBuilderBase class provides abstract declaration of the GridBuilder interface. This interface offers the Create() method that creates a working copy of the prototype grid builder around a specified GridBindingManager.

DetailGridViewBuilderBase

This class provides an abstract interface definition for builders that create drill-down detail views linked to a parent grid view (for control suites that support drill-down detail views). This is not the same as a standard GridBuilder; this class actually *creates* a view inhabited by a grid and populates and styles it based on the data source of the parent grid.

GridBuilderColumnOptions

This class provides control suite-independent options for building a column in a data grid view. It is used to encapsulate parameters to the methods of grid builder. The options include:

- The entity property name to be bound to the column
- The data converter to be used
- The read only status of the column
- The title (header text) of the column
- The width of the column

3.3.2 Grid Builder Services

The GridBuilder service supplied by IdeaBlade/CAB provides a registry of GridBuilders. Registered GridBuilders are indexed and may be accessed by name and by bound entity type.

Applications and modules typically create prototype grid builders at startup or module load time and register them with the grid builder service.

GridBuilderServiceBase

The GridBuilderServiceBase class provides abstract declaration of the GridBuilderService interface, as well as static methods to access prototype GridBuilders (by name or bound type) from a specified WorkItem.

GridBuilderService

This is the concrete implementation of the interface defined by GridBuilderServiceBase. It maintains a dictionary of prototype GridBuilder objects (typically populated by the application at startup, or by modules at load time), indexed by name as well as the bound entity type. An application can create a specialized GridBuilderService populated (by its constructor) with entity-specific grid builders.

GridColumnWidthsServiceBase

This is an example of a simple service implementation. It is typically added to the root WorkItem by the application at startup and provides default values for data grid column widths based on their data types. An application may define a child class that overrides or extends these values, then adds the child service to a WorkItem to make it available to grid builders.

3.3.3 Control Suite Specific Grid Builder Classes

Much of the functionality of a grid builder depends on the control suite supplying the underlying data grid control. Accordingly, IdeaBlade/CAB supplies control-suite-specific grid builder support classes in dedicated assemblies. These share a common class structure:

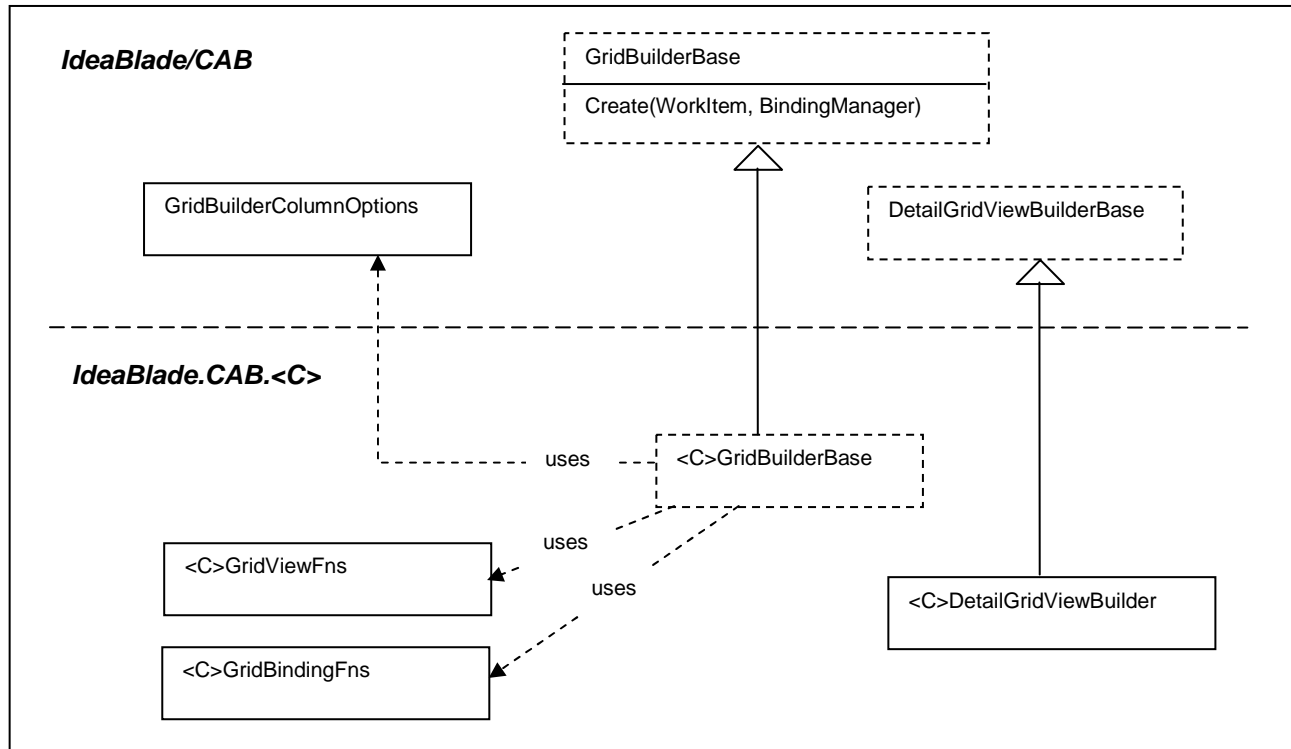


Figure 15: Control Suite Specific Grid Builder Class Structure

<C>GridBuilderBase

These classes provide base functionality, including basic remove/hide column methods for grid builders targeted at a specific control suite’s data grid control. They reference suite-specific visual and DevForce components, including the suite’s data grid control and suite-specific GridBindingManager, GridBindingDescriptor and GridBindingDescriptorCollection classes provided by DevForce.

<C>DetailGridViewBuilder

This class provides a concrete implementation of DetailGridViewBuilderBase for a specific control suite that supports drill-down detail grids.

<C>GridViewFns

This static class provides global functions that assist in styling a data grid for a particular control suite. It provides properties that supply “standard” styles for cells, rows, alternate rows, etc. as well as a method that applies these standard styles to an entire data grid control.

<C>GridBindingFns

This static class provides additional global functions that assist in styling a data grid for a particular control suite to display a row state image.

3.3.4 Default Grid Builder Implementation Selection

As with other control-suite-specific implementations, the application foundation skeleton declares the AppGridBuilderBase class that is “switched” using conditional assembly to inherit from the “primary” control suite. This simplifies the task of creating new grid builders for an application.

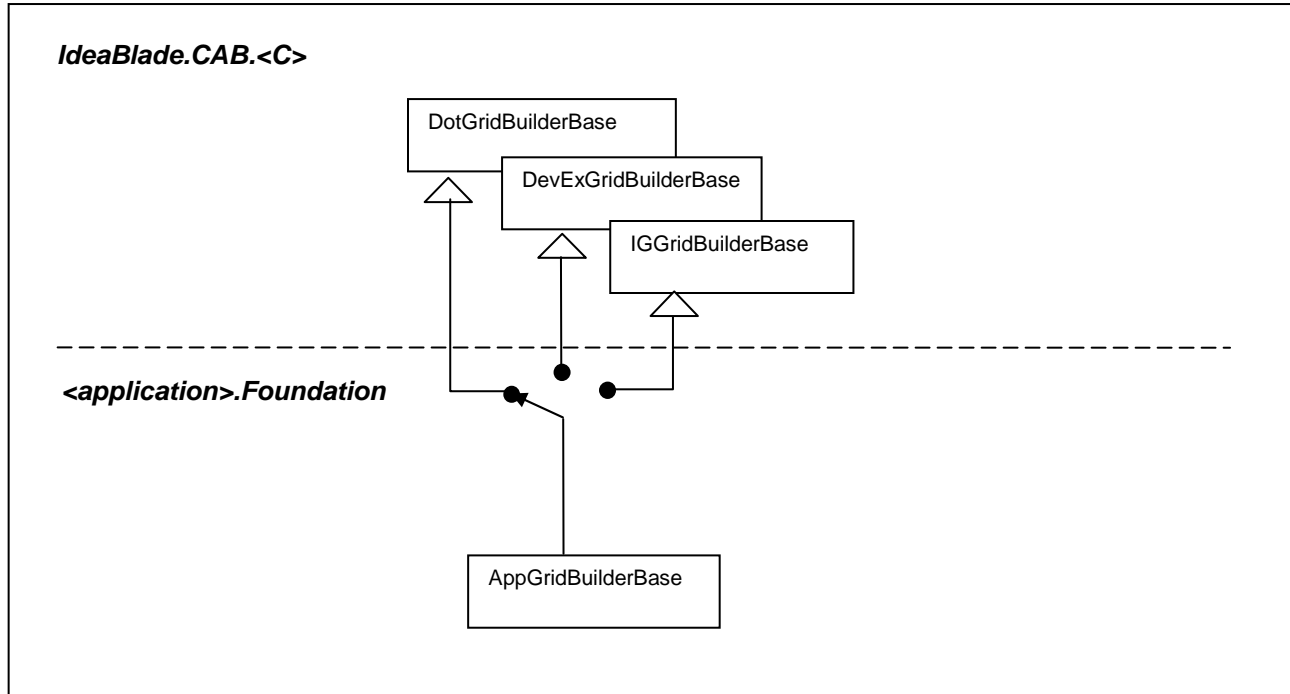


Figure 16: Selection of Primary Control suite specific GridBuilderBase Implementation

3.3.5 Tab View Controllers

Layout views containing tab views pose unique problems. A tab paradigm is often used to display multiple aspects of an entity or other bound object. As the user navigates to different bound objects, the information displayed on separate tabs must be kept “in sync”. IdeaBlade/CAB provides page and WorkItem controllers with helper classes that assist in implementing this logic.

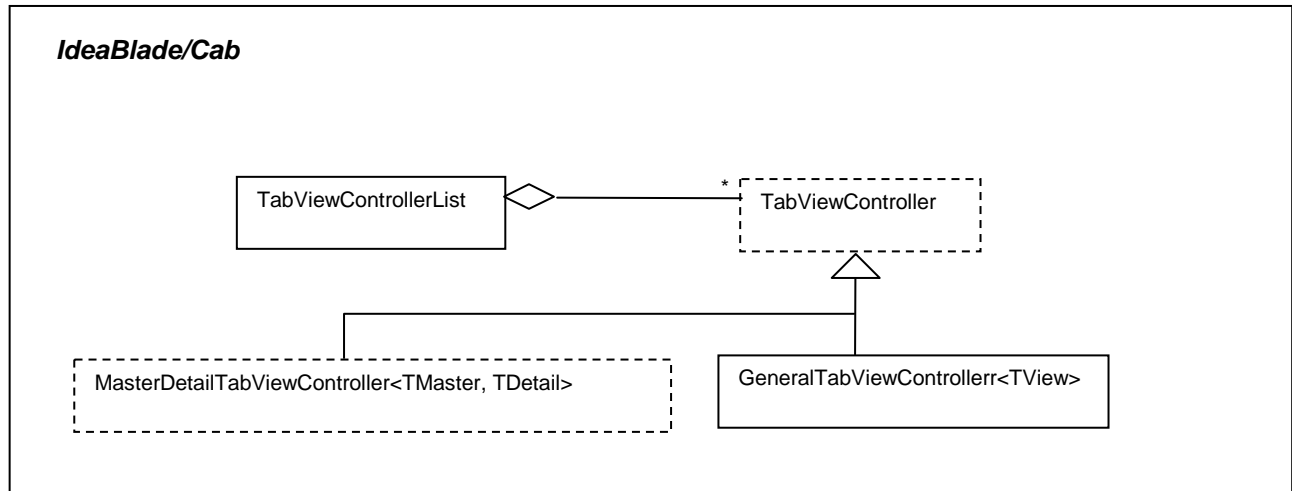


Figure 17: Tab View Controller Class Structure

TabViewController

This abstract class provides the basic interface to a `TabViewController`. It supplies some basic functionality, including access to useful services from its owning `WorkItem`. These include the `Editor Service`, `MessageBoxService` and the `ViewFactoryService`.

`TabViewControllers` create views to be displayed in Tab workspaces by executing a series of phases:

1. Create the view and set its `SmartPartInfo` into the containing tab page.
2. Set one or more binding sources for the view. Often a tab workspace will hold multiple views, all bound to the same parent binding source. The `TabViewController` allows specification of an `EntityManager` for the view to assist in keeping the view synchronized with other views.
3. Set up view-specific events, often responding to events on the binding source.
4. Add the view to its containing tab workspace.
5. When necessary, refresh the view from its binding source.

`TabViewController`, though abstract, supplies a significant amount of default functionality.

GeneralTabViewController

The `GeneralTabViewController` is used to create and manage a single view bound to the main binding source to be displayed in a tab workspace. It supplies simple implementations of the abstract methods of `TabViewController`.

MasterDetailTabViewController

IdeaBlade/CAB provides extensive support for displaying and editing information in master-detail form. The `MasterDetailTabViewController` class provides control logic for the `MasterDetail` layout view and also supports including the resulting view in a tab workspace.

TabViewControllerList

This class maintains a list of the `TabViewControllers` associated with a tab workspace. It implements methods for the view-construction phases mentioned above. Each of these methods invokes the corresponding method on each of the builders in the `TabViewControllerList`.

Typically, a `WorkItem` controller creates a `TabViewControllerList`, then populates it with controllers for the views to be displayed in a tab workspace.

3.4 Workspaces

IdeaBlade/CAB defines several common workspaces useful for applications:

3.4.1 CABWindowWorkspace

This class, generated by SCSF, implements a workspace that shows each `SmartPart` in a separate window. It keeps track of windows and their activation.

3.4.2 WindowWorkspace

Adds functionality to `CABWindowWorkspace` to display the windows on top of the parent window if there is one.

3.5 Services

IdeaBlade/CAB supplies a number of services useful to applications. Many of these are described elsewhere in this document.

DefaultResourceService

DevForceLoggingService

EntityEditorService

GridBuilderService

ListConverterService

MessageBoxService

NavBarService

PersistenceServiceErrorHandlerService

StopWatchLoggerService

ViewFactoryService

EntityManagerService

3.6 WorkItem Controllers

The IdeaBlade/CAB framework provides support for composite views displaying `DevForce` entities.

3.6.1 Entity View Controllers

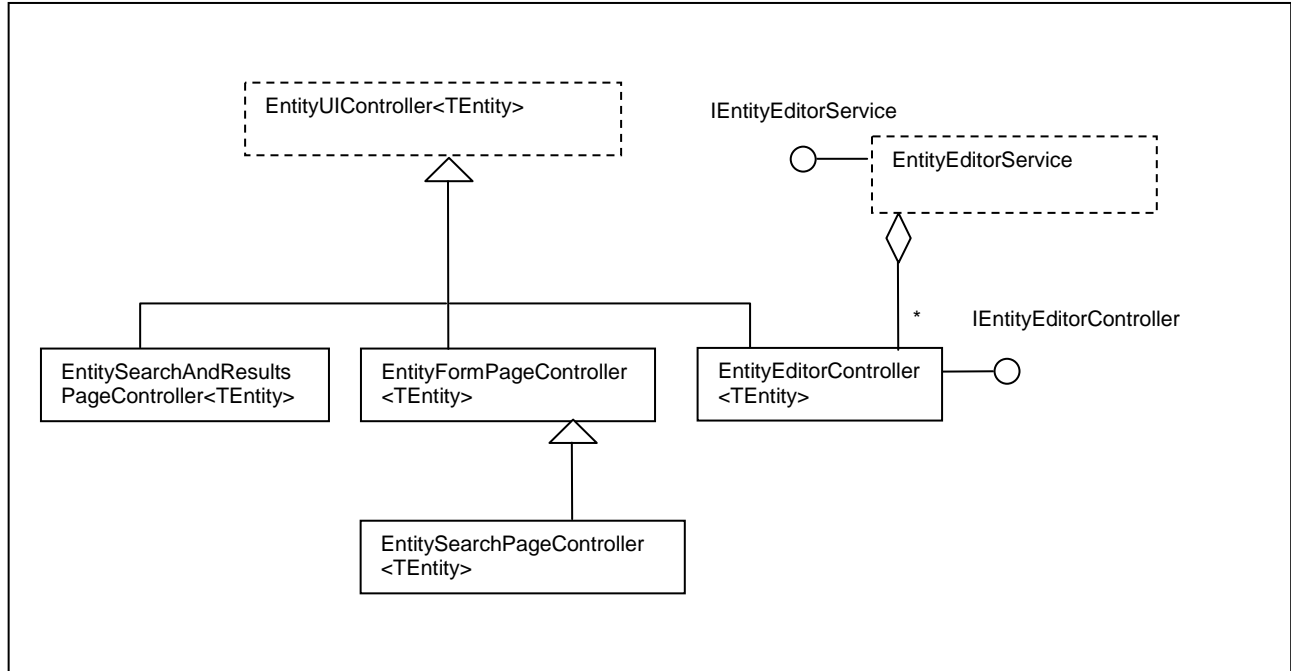


Figure 18: Entity View Controllers Class Structure

EntityUIController

The EntityUIController is an abstract class that provides basic functionality for views of a particular root entity type. It provides:

- A static ShowPage() mechanism.
- Support for injection of necessary services by the object builder
- Lifecycle management, including creation of an EntityManager for local synchronization with changes in other views.
- Access to the main view, workspace and binding source.
- Access to the currently displayed root entity.

EntityFormPageController

Controls a "Form Page" displayed within a Workspace for pages. A "Form Page" presents a single entity with a summary of the entity at the top and tabs below, each tab presenting details about the entity.

EntitySearchPageController

Controls a "Query Page" displayed within a Workspace for pages. A "Query Form Page" presents a page divided into two partitions. The top part holds an entity Searcher and Searcher query results grid. The bottom holds the current entity in the standard "Page View" displayed as a summary of the entity over tabs, each tab presenting details about the entity.

EntitySearchAndResultsPageController

Controls a "Grid Page" displayed within a Workspace for pages. A "Grid Page" presents a page divided into two partitions. The top part holds an entity Search View. The bottom holds the editable Search Results displayed as a grid.

EntityEditorController

Builds and manages an Editor for a specific root entity type.

Edits a single root entity within a separate IEntityManager and PersistenceManager sandbox. The editor is known by the entity it edits; additional requests to IEntityEditorService for an editor for the same entity type will return the same editor.

The editor displays within a non-modal WindowWorkspace. The editor control and its parts - WorkItem, views, etc. - are disposed when the window closes.

Never directly "new" an EntityEditor; it should always be created via a service that implements IEntityEditorService.

EntityEditorService

The EntityEditorService provides and manages editor views for DevForce entities. It allows a single editor to be used for an entity, even if it is requested and accessed from multiple points in the application. Once an editor for an entity (called the root entity since the editor might edit child entities as well) is created and supplied by the EntityEditorService, it may be requested in several ways:

- By supplying the entity to be edited itself.
- By supplying the primary key of the root entity of the editor.
- By supplying the editor's Id (typically the id of the editor's WorkItem).

The entity editor service handles creation of a WorkItem for the editor and display of the editor in the applications "Window Workspace" which presents the editor view in a dedicated modal window.

3.7 Entity Management

The entity management classes work with the IdeaBlade DevForce object-relational mapping classes to provide access to one or more object models from elements of a CAB-based design. Working together, EntityManagers and their parent EntityManagerService provide for local (i.e., within the client-side application) synchronization of the entity caches of multiple persistence managers.

Each EntityManager instance wraps a single DevForce persistence manager but multiple entity managers may refer to the same persistence manager. An application (including dynamically loaded modules) may create and use multiple EntityManagers, themselves managed by the EntityManagerService. In addition, the EntityManagerService supports caching of schema information for a persistence manager, enabling future persistence managers targeted at the same persistence server to be created and populated without multiple round trips to the server.

EntityManager

The EntityManager class is a wrapper for the DevForce PersistenceManager class. It provides support for local (i.e., client-side) synchronization of entities in multiple persistence managers. Using unique marker ids to track outstanding asynchronous queries, it captures entities that have been modified and saved by its persistence manager. These are made available to other EntityManager instances to allow them to keep their persistence manager caches in sync.

The EntityManager uses a strategy pattern to implement merging of results of save operations.

In concert with the EntityManagerService, an EntityManager caches schema information to allow rapid initialization of new persistence managers

EntityManagerService

The EntityManagerService creates and supplies new EntityManager instances on demand. Multiple constructor overloads allow wrapping of an exiting persistence manager, creating and initializing a copy from the schema cache, or initializing from the underlying data source.

The service maintains indices of entity managers both by id and persistence manager for rapid access. It exposes the IEntityManagerParent interface to created EntityManagers to provide access to the schema-caching functions.

EntitySchemaCache

The EntitySchemaCache class manages a cached copy of all or part of the schema of a data source for initialization of new persistence managers. It maintains a local persistence manager populated with the cached entity types. When required, retrieves the EntitySet from the local persistence manager and “restores” it to a newly-created persistence manager.

EntitySaveResult

The EntitySaveResult class is used to captures the result of a save operation by an entity manager, including success/failure and any exception from the persistence manager. It holds separate lists of saved and deleted entities and supplies (and caches) dictionaries keyed by entity type on demand.

Instances of EntitySaveResult are created by EntitySaveStrategy implementations. They are used by EntitySaveResultMergeStrategy implementations to synchronize other EntityManagers based on the same data source.

EntityManagerSaveStrategy

An implementation of EntityManagerSaveStrategy captures the result of a save operation on an EntityManager in an EntitySaveResult instance. It may applies various rules to determine those entities whose saving/deletion is to be synchronized with other EntityManagers. This class provides a default implementation, but its behavior may be specialized by application-specific child classes.

EntityManagerSaveResultMergeStrategy

An implementation of this class provides the strategy for merging the results of a save operation on one EntityManager into other EntityManagers on the same data source. It allows its owning EntityManager (the merge target) to customize its behavior to some extent and provides a delegate for post-merge processing. It provides a default implementation, but its behavior may be specialized by application-specific child classes.

AsyncMarkerManager

This class maintains collection of guids uses to track outstanding asynchronous queries.

CacheSavingEntityManager

This is an abstract implementation of an EntityManager that can save the contents of its persistence manager to local disk. Do not confuse saving of persistence manager cache with entity schema caching by EntityManagerService. Appears to not be complete (or at least not fully-tested)

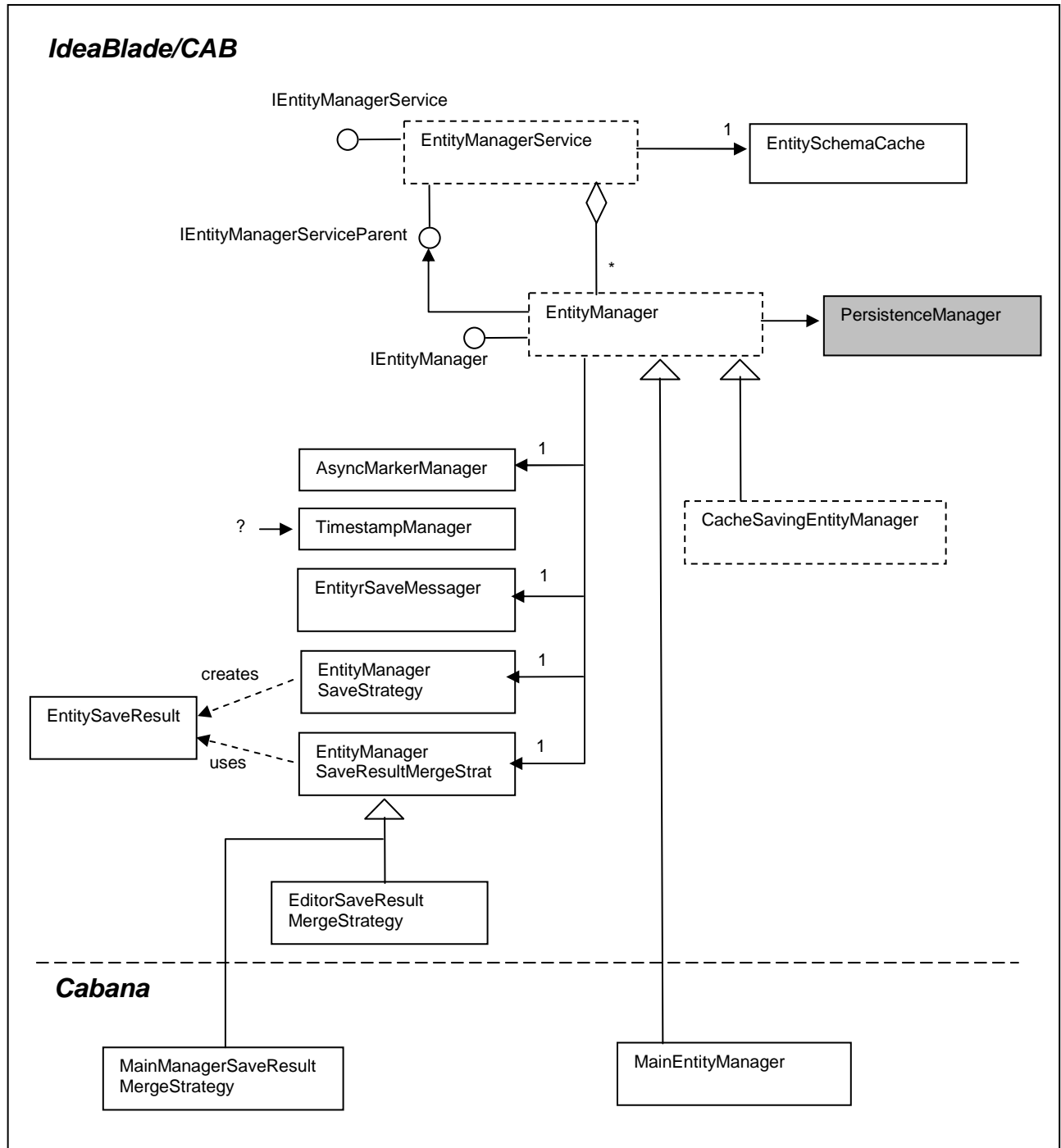


Figure 19: Entity Manager Class Structure

4 Skeleton Application

The application shell provides the basic startup and configuration structure for an application based on IdeaBlade/CAB. The ShellApplication and ShellForm classes are configured to inherit from the appropriate suite-specific base class using conditional compilation.

4.1 Application Shell

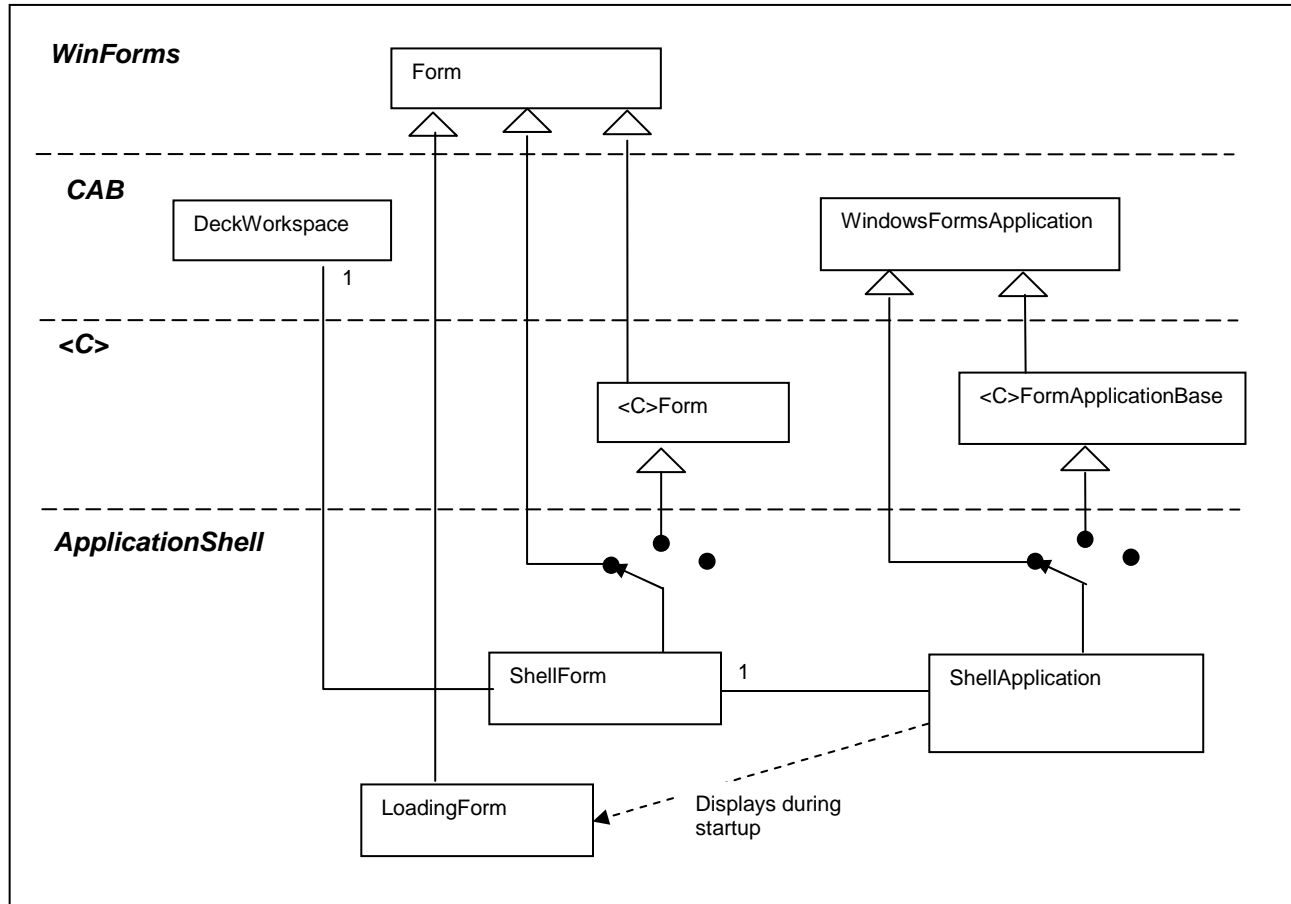


Figure 20: Skeleton Application Class Structure

ShellApplication

The ShellApplication class contains the static Main() method of the application and is responsible for establishing the CAB-based environment for the application. It is “switched” using conditional compilation to inherit from a third party control suite base class and to perform any necessary control suite initializations. It performs a number of initialization functions:

- Displays the Loading form on a separate thread during initialization.
- Provides for polite handling of exceptions during initialization.

- Initializes common services needed by all IdeaBlade/CAB applications plus any application-specific services.
- Sets up the CAB visualizer if not running in remote mode
- Initializes general WinForms parameters.
- Initializes parameters for any third-party control suite.
- Initializes DevForce, and handles login to the persistence server
- Runs the application (starts the message pump) using ShellForm as the top level form.

ShellForm

ShellForm is the top level visual component of the application. It is “switched” to inherit from a third party control suite base class. It contains a single DeckWorkspace with the well-known name “ShellLayout”.

Loading Form

A simple graphic background with a rotating marquee progress bar displayed during initialization.

4.2 Configured Services

The skeleton application shell configures IdeaBlade/CAB services into the root WorkItem. The configured services include:

4.2.1 DevForce Services

DevForceLoggingService	Provides logging services...
BaseViewFactoryService	Provides a source for control suite-specific views.

4.2.2 SCSF Services

ProfileCatalogModuleInfoStore	Implements IModuleInfoStore
WorkspaceLocatorService	Implements IWorkspaceLocatorService
XmlStreamDependentModuleEnumerator	Implements IModuleEnumerator
DependentModuleLoaderService	Implements IModuleLoaderService
ActionCatalogService	Implements IActionCatalogService
EntityTranslatorService	Implements IEntityTranslatorService

5 The Cabana Reference Application

The Cabana reference application is a WinForms-based CAB application. It uses the IdeaBlade CAB extensions for back-end data management and data binding, the Developer Express DevX UI control suite.

Like many CAB-based applications, the Cabana reference application includes common interfaces, components and facilities used by multiple modules. The “interface” assembly defines the interfaces to

these common facilities and is referenced by all task-specific modules. The “foundation” assembly defines the foundation module that provides the implementation of these facilities.

5.1 Common Interface Elements

5.1.1 Name Constant Definitions

The Cabana reference application extends the names defined by SCSF and the IdeaBlade/CAB extensions with application-specific names:

CommandNames

```
ShowSplashPage = "ShowSplashPageCommand";
```

ResourceNames

Image Resource Names

```
AdminImage = "AdminImage"
CustomersImage = "CustomersImage";
DefaultSalesRepImage = "DefaultSalesRepImage"
DevForceLogoImage = "DevForceLogoImage"
SalesOrdersImage = "SalesOrdersImage"
SalesRepsImage = "SalesRepsImage"
ShippingImage = "ShippingImage"
SplashImage = "SplashImage"
SonOfSplashImage = "SonOfSplashImage"
SplashTaskImage = "Splash1Image"
SonOfSplashTaskImage = "Splash2Image"
UserImage = "UserImage"
```

Text Resource Names

```
ApplicationTitleText = "ApplicationTitle";
CollapseSearchText = "CollapseSearch";
ExpandSearchText = "ExpandSearch";
```

ViewNames

```
CustomerPage = "CustomerPageView";
EmployeePage = "EmployeePageView";
SalesRepPage = "SalesRepPageView";
SecurityGroupPage = "SecurityGroupPage";
SupplierPage = "SupplierPageView";
SplashPage = "SplashPageView";
UserPage = "UserPageView";
```

WorkspaceNames

```
SplashWorkspace = ShellContent;
```

In addition the Cabana application defines an additional category of constants:

ListConverterNames

```
Manager = "ManagerListConverter";
ManagerId = "ManagerIdListConverter";
OrderCustomer = "OrderCustomerListConverter";
OrderCustomerId = "OrderCustomerIdListConverter";
```

```
SalesRep           = "SalesRepListConverter";  
SalesRepId        = "SalesRepIdListConverter";  
Shipper           = "ShipperListConverter";  
ShipperId         = "ShipperIdListConverter";  
TitleOfCourtesy  = "TitleOfCourtesyListConverter";
```

5.1.2 Interfaces

5.1.2.1 IOrderEditorController

This is a “marker” interface to facilitate retrieving order editors from the EditorService.

5.1.2.2 IVerifierResultsAlertsView

This is planned to move into the common views supplied by IdeaBlade/CAB.

5.2 Common Infrastructure Components

5.2.1 Services

ListConverterService

Specializes the ListConverterService to provide list converters for application-specific entities:

```
ManagerListConverter  
ManagerIdListConverter  
OrderCustomerListConverter  
OrderCustomerIdListConverter  
SalesRepListConverter  
SalesRepIdListConverter  
ShipperListConverter  
ShipperIdListConverter  
TitleOfCourtesyConverter
```

ResourceService

Specializes DefaultResourceService to provide application-specific resources:

```
ConditionImageTransparentColor  
LogoImage  
NavImageTransparentColor
```

5.2.2 Grid Builders

The Cabana reference application defines specialized grid builders for its application-specified entities and populates the GridBuilderService supplied by IdeaBlade/CAB with prototype implementations. Application-specific grid builders are created for the following entities:

```
User  
SecurityGroup  
SalesRep  
Customer  
CustomerOrder  
Order
```

```
OrderDetail  
Supplier
```

All of the application-specific grid builders inherit from AppGridBuilder which defines the “primary” control suite used by the application.

5.2.3 Tab View Controllers

The Cabana reference application defines an application-specific Tab View Controller to assist in managing the OrderMasterDetailView

```
OrderMasterDetailTabViewBuilder
```

5.2.4 Menu Extensions

Extends the applications main menu with necessary items.

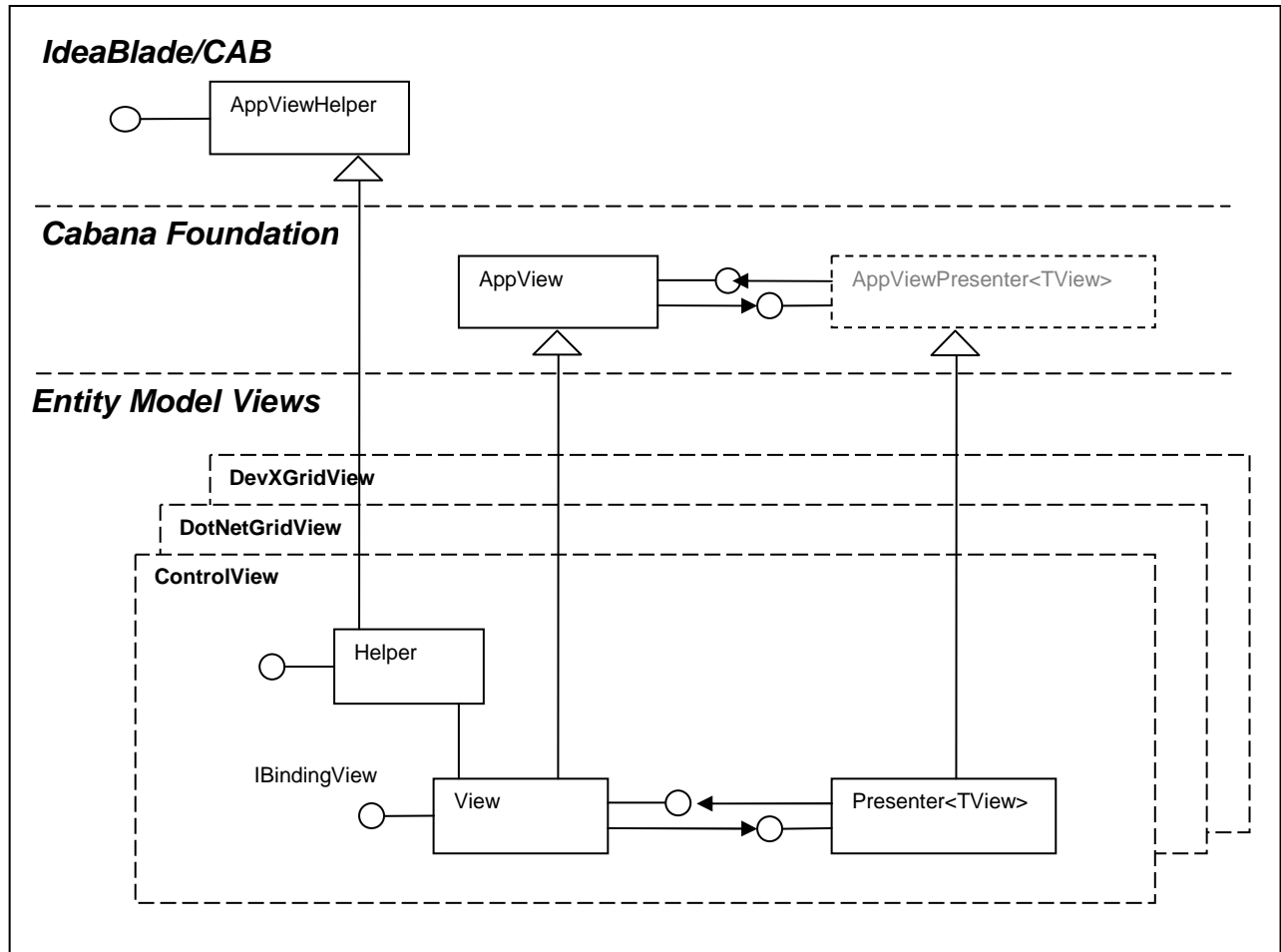
Doesn't appear to be quite done.

5.2.5 Verifiers Setup

Not done yet.

5.2.6 Entity Model Base Views

The Cabana Foundation provides several base classes for views supporting the IdeaBlade DevForce framework. These implement IBindingView which is used by common presenters.



5.3 Cabana Foundation Module

The Cabana reference application extends the skeleton application foundation module to set up common facilities used by multiple business modules within the Cabana foundation.

5.3.1 IdeaBlade/CAB Services

The Cabana foundation module configures several services defined by IdeaBlade/CAB into the root WorkItem:

MessageBoxService	Displays a message to the user in a configured and localized message box. Provides automatic marshalling to the UI thread. Provides a central point for customization of how messages are displayed to the user.
StopWatchLoggerService	Provides logging of performance measurement information to...
PersistenceServerErrorHandlerService	Handles errors from the DevForce persistence server.
NavBarService	Handles addition of new tasks to the navigation bar, deferred until all configured modules have been loaded to allow sorting by display order.

EntityManagerService	Supplies EntityManager instances. Initially configured to support the main persistence manager.
EntityEditorService	Supplies editors customized for application-specific entities.
GridColumnWidthsServiceBase	Supplies grid column widths for various data types to grid builders.
ViewFactoryService	Configures the ViewFactoryServiceBase (added to the root WorkItem by the application shell) to provide application-specific views.
GridBuilderService	Supplies application-entity-specific grid builders.

5.3.2 Cabana-Specific Services

In addition to the IdeaBlade/CAB defined services, the Cabana foundation module configures two services defined by the Cabana reference application:

ListConverterService	Supplies list converters for application-specific entities.
ResourceService	Specializes the ResourceService to provide application-specific resources

5.3.3 Shell Display Setup

The Cabana foundation module performs several display setup functions:

- Obtain an instance of the ShellLayout view (defined by IdeaBlade/CAB) from the ViewFactoryService and add it to the ShellLayout workspace defined by the application template. The ShellLayout view in turn defines the Navigation and Content workspaces used all other modules.
- Locate the top-level ShellForm of the application and configure it with logo and title text information.
- Creates a WindowWorkspace for display of modal pop-up windows. Set the ShellForm as its parent form.
- Create a WindowWorkspace for display of non-modal editor windows and set the ShellForm as its parent.
- Set the ShellForm as the parent of message box windows to be supplied by the MessageBoxService.

5.3.4 Navigation Bar Setup

The Cabana foundation module subscribes to the RunStarted event of the root WorkItem. When this event is fired, all modules have been loaded and the foundation module requests the NavBarService to show all navigation bars that have been queued by other modules. This allows the bars to be sorted and added in the correct display order.

5.3.5 Command Handlers

The Cabana foundation module provides handlers for global commands:

- HelpAbout
- ApplicationExit

5.4 Cabana Task Modules

5.4.1 Splash

5.4.2 Admin

5.4.3 SharedEditor

5.4.4 SalesRep

SalesRepModuleInit

Creates a new WorkItem controlled by a SalesRepModuleController, then invokes the controller's Run method.

SalesRepModuleController

Installs a module-specific resource service to provide resources specific to this module. Subscribes to the ShowPage event of the SalesRepPage view.

SalesRepSearchPageController

This is the controller for the SalesRep page. It inherits from the EntitySearchPageController<entity> class provided by IdeaBlade/CAB and overrides three initialization methods:

- Overrides CreateSearchView() to add a SimpleSearcherView to the “Search” SmartPartPlaceholder.
- Overrides CreateSummaryView() to add a new SalesRepSummaryView to the “SummarySmartPartPlaceholder”.
- Overrides CreateSearchResultsView() to add a grid view of the Employee entity. (This is actually the default behavior of EntitySearchPageController<Employee>).
- Creates and configures a new SalesRepSearcher
- Creates TabViewBuilders for detail views:
 - General SalesRep information
 - Sales Rep Orders
 - Sales Rep Customers

5.4.5 SalesRepNav

5.4.6 SalesOrder

A1 Appendix 1: Global Definitions

A1.1 Global Name Definitions

CommandNames

ApplicationExitCommand
AddNewCommand
DeleteCommand
EditCommand
SaveCommand
SaveAndCloseCommand
CancelAndCloseCommand

EntityManagerNames

MainEntityManager

EventTopicNames

StatusUpdateEvent Causes shell to update the status panel
ExitApplicationEvent
VerifierResultsChangedEvent
SetEditorAlertsVisibilityEvent
ResetBindingsEvent
SuspendApplicationEvent
ResumeApplicationEvent

ResourceNames

Logo Images

LogoIcon
LogoImage

Condition Images

CriticalConditionImage
InformationConditionImage
WarningConditionImage
ConditionImageTransparentColor

Operation Images

AddNewImage
EditImage
DeleteImage
FindImage
RefreshDocImage
SaveImage
UndoImage

RowState Images

RowStateAddedImage
RowStateDeletedImage
RowStateModifiedImage
RowStateUnchangedImage

Texts

```
EditText      = "Edit";  
AddNewText    = "AddNew";  
DeleteText    = "Delete";  
SaveText      = "Save";
```

UIExtensionSiteNames

```
MainMenu      = "MainMenuSite"  
MainToolBar   = "MainToolBarSite"  
MainStatus    = "MainStatusSite"  
GridViewToolStrip = "GridViewToolStripSite"
```

WorkspaceNames

```
ModalWindows  = "ModalWindowsWorkspace"  
ShellLayout   = "ShellLayoutWorkspace"  
ShellNavigation = "ShellNavigationWorkspace"  
ShellContent  = "ShellContentWorkspace"
```