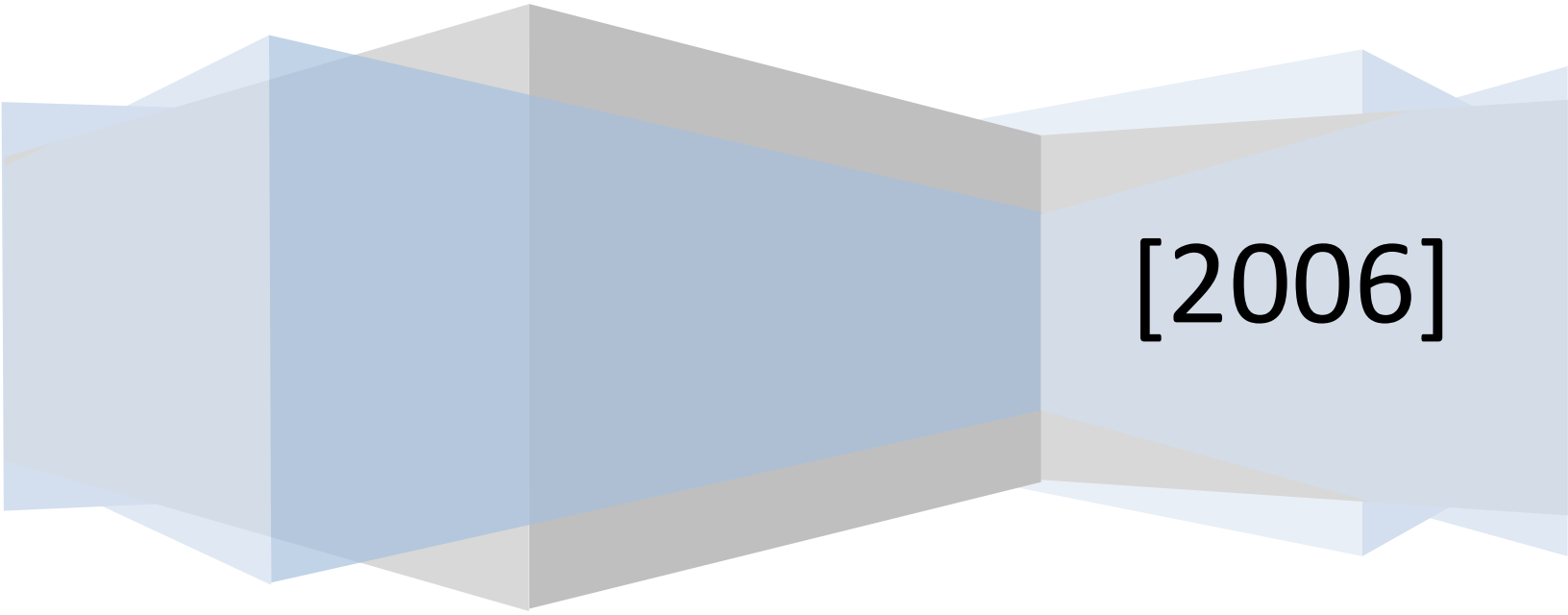
Microsoft

Designing Smart Clients Based on CAB and SCSF

Architectural Guidance for Composite Smart Clients

Mario Szpuszta
architect | microsoft Austria
email: marioszp@microsoft.com



[2006]

Contents

Introduction

This white paper, originally created for RACON Software GmbH, which is a Software house of the Raiffeisen Banking Group in Upper Austria and provides architectural guidance for designing and implementing composite smart clients based on the Composite UI Application Block and the Smart Client Software Factory provided by the Microsoft patterns & practices group.

The original intention of this white paper was to support the architects and lead developers of RACON Software GmbH for designing and implementing a new bank desktop composite client application. The primary idea of the bank desktop application found its origin in the fact that the banks-institutes have many different client-based applications to support bank employees in their daily business. Primarily, the problem was that each of these applications worked differently, and the employees struggled with the different user experiences of these applications. Furthermore, the applications are not integrated; this results in many tasks being accomplished multiple times in different applications. From the IT-perspective, managing that many applications is fairly hard and the reuse of common functionality is nearly impossible. Therefore, RACON Software GmbH had the idea of creating a common application as a foundation for all bank applications. Each and every bank application should be hosted in this common bank application. The bank applications, which finally are modules dynamically plugged into the common bank application, should be loaded and initialized on demand based on the user's security context so users see only the parts they are allowed to work with. Of course, a common infrastructure such as a bank desktop can also manage common services used by all the different applications that are dynamically plugged in to a central infrastructure. This makes management easier and allows re-use of common client-side services. Because the modules are dynamically configurable, one of the core requirements was to enable the components to communicate in a loosely coupled way so the modules do not really depend on each other. Finally, the idea of the bank desktop composite application was developed. It provides a common client-application infrastructure for every bank employee of the bank institutes. Each client-based application will be integrated into this bank desktop common application by a role-based configuration suited to the tasks of an employee of a specific role. Therefore, the bank desktop is a very good example of a very special type of smart client: *composite smart clients*. The offerings from Microsoft for designing and developing composite smart clients is made up of the Composite UI Application Block and Smart Client Software Factory. This article is all about these technologies.

Scope of This White Paper

The primary goal of this white paper is to provide architectural guidance for developing composite smart clients based on Composite UI Application Block (CAB) and the Smart Client Software Factory (SCSF). Therefore, this white paper helps architects with the following decisions:

- Whether to structure project teams as CAB and SCSF, which affects the development process of smart clients
- How to identify the primary components for composite smart clients
- How to package these components into deployment units
- How to use infrastructure components, such as the event broker and commands
- How to determine the design of the Shell

The second part of the white paper contains checklists that include instructions for using CAB and SCSF. These instructions can be provided to developers for a faster ramp-up time when they start working with both CAB and SCSF. This white paper is **neither a detailed documentation on the APIs of Composite UI Application Block nor a detailed description on the guidance automation packages provided by the Smart Client Software Factory**. For more information about CAB and SCSF, see the following resources on MSDN:

- "Smart Client – Composite UI Application Block": http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/cab.asp
- "Smart Client Software Factory": http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/scsflp.asp
- "New Guidance and Tools for Building Integrated Desktop Applications": http://msdn.microsoft.com/msdnmag/issues/06/09/SmartClients/

Requirements for Reading This White Paper

To completely understand this white paper, you need to know about the basic structure of the Composite UI Application Block. Understanding the Smart Client Software Factory is optional for the first part of the white paper and is required for the second part. For detailed information about CAB and SCSF, see the links provided in the section, “Scope of this White Paper.”

To test and follow the samples introduced in this white paper, you need to install the following components on your development computer:

- .NET Framework 2.0: http://www.microsoft.com/downloads/details.aspx?FamilyID=0856eacb-4362-4b0d-8edd-aab15c5e04f5&DisplayLang=en
- Visual Studio 2005 Professional Edition or a later version: http://msdn.microsoft.com/vstudio/products/vspro/default.aspx
- Enterprise Library for .NET Framework 2.0 – January 2006: http://www.microsoft.com/downloads/details.aspx?FamilyId=5A14E870-406B-4F2A-B723-97BA84AE80B5&displaylang=en

- Composite UI Application Block in C#: http://www.microsoft.com/downloads/details.aspx?FamilyId=7B9BA1A7-DD6D-4144-8AC6-DF88223AEE19&displaylang=en
- Guidance Automation Extensions: http://www.microsoft.com/downloads/details.aspx?familyid=C0A394C0-5EEB-47C4-9F7B-71E51866A7ED&displaylang=en
- Guidance Automation Toolkit: http://www.microsoft.com/downloads/details.aspx?familyid=E3D101DB-6EE1-4EC5-884E-97B27E49EAAE&displaylang=en
- Smart Client Software Factory – June 2006: http://www.microsoft.com/downloads/details.aspx?FamilyId=2B6A10F9-8410-4F13-AD53-05A202FBDB63&displaylang=en

Part I: Architectural Guidance for Composite Smart Clients

The first part of this white paper covers architectural guidance for creating composite smart clients based on the Composite UI Application Block and Smart Client Software Factory. This part is aimed at providing guidance to help you identify and structure the components of your composite smart client and find a way to package them.

Terms and Definitions

Before you get to the details of the guidelines provided by this first part, you will need to make sure to understand basic terms that are used throughout this white paper. This section summarizes these terms.

Composite UI Application Block (CAB)

The Composite UI Application Block is a ready-to-use building block provided by the Microsoft patterns & practices group for creating complex, modularized smart client applications. This building block basically provides the following functionality:

- Dynamically loading independent yet cooperating modules into a common shell
- Event broker for loosely coupled communication between modules and parts of modules
- Ready-to-use Command pattern implementation
- Base classes for Model-View-Control (MVC) pattern implementations
- Framework for pluggable infrastructure services ranging, such as authentication services, authorization services, module location, and module loading services

CAB is built for designing applications based on a use case–driven approach by clearly separating use case controller classes from views with their controller classes or presenter classes. Central client-side services can be added through the services infrastructure provided by CAB. Table 1 summarizes the major terms related to CAB that are important for architectural decisions.

Table 1: Important Terms and Definitions for Composite UI Application Block

Class / Term	Description, Purpose
WorkItem	Seen technically, a WorkItem is just a container for managing objects, which are required for accomplishing a certain task. Such objects are state objects, views and their presenters/controllers, or commands for launching an action. At a more high-level view, a WorkItem is a class encapsulating logic for one dedicated use case. It can be seen as a *use case controller* that knows all the different aspects of a use case. As such, a WorkItem holds state for all parts involved in a use case, such as views necessary to display, or sub-use cases. Furthermore, it acts as an entry point into the use case (or one of its sub-use cases) it is responsible for.
Service	Services encapsulate functionality that is common for the whole client application, for a specific module, or just for WorkItems within a certain hierarchy of WorkItems (for example, for sub-WorkItems of a specific parent WorkItem). Typical services are security services responsible for authentication or authorization or Web service agents encapsulating service communication

	and offline capability.
Module	Multiple, logically related WorkItems can be summarized into a single unit of deployment. A module is an example of a unit of deployment. Configuration of CAB-based smart clients basically works on a module-level. Therefore, finding the right granularity for encapsulating WorkItems into modules is crucial.
ProfileCatalog	The profile catalog is just a configuration that specifies which modules and services need to be loaded into the application. By default, the catalog is just an XML file that resides in the application directory. This XML file specifies which modules need to be loaded. By writing and registering a custom **IModuleEnumerator** class, you can override this behavior to get the catalog from another location, such as a Web service.
ModuleLoader	This is a general service provided by CAB, which is responsible for loading all the modules specified in a profile catalog. It just walks through the configuration of the profile catalog, tries to dynamically load the assembly specified in a catalog entry, and then tries to find a **ModuleInit** class within that assembly.
ModuleInit	Each module consists of a **ModuleInit** class, which is responsible for loading all the services and WorkItems of a module. (In addition to services and WorkItems, the class is also responsible for aspects, such as user interface extensions to the shell; for more information, see the description for UIExtensionSite later in this table.)
Shell application	The Shell application is the primary application that is responsible for initializing the application. It is responsible for dynamically loading modules based on the configuration and launching the base services for the smart client application, in addition to starting the main form (Shell).
Shell	This is the main form of the application; it provides the user interface that is common to all dynamically loaded modules. The Shell always hosts the root WorkItem, which is the entry point into any other parts of the application, such as services, modules, and WorkItems that are created and registered by these modules.
Workspace	A workspace is a control that is primarily responsible for holding and displaying user interface elements created by WorkItems. Usually workspaces are added to the Shell (or to other extensible views within your WorkItem hierarchy) and act as containers for parts of the UI that are going to be dynamically filled with UI items that are provided by WorkItems or sub-WorkItems. Any user control can be a workspace by implementing the **IWorkspace** interface. Out-of-the-box, CAB includes classic container controls as workspaces, such as a tabbed workspace.
UIExtensionSite	UIExtensionSites are special placeholders for extending fixed parts of the Shell such as menus, tool strips or status strips. Unlike workspaces, they are intended to be used for parts of the UI, which should not be completely overridden by WorkItems; instead, they should just be extended (such as adding a new menu entry).
Command	Command is a base class of CAB for implementing the Command pattern. CAB supports classic commands created manually or declarative commands by applying the [**CommandHandler**] attributes to the method; these attributes act as command handlers. You can register multiple invokers for one command

	(such as click events of multiple controls).
EventPublication	**EventPublications** are used by event publishers for loosely coupled events. An event publisher implements the event through .NET Framework events that are marked with the [**EventPublication**] attribute. Events are uniquely identified by event URIs (unique strings). Only subscribers using subscriptions with the same event URI will be notified by CAB. A subscriber needs to have a method with the same signature as the event used [EventPublication] and needs to be marked with the [**EventSubscription**] attribute.
EventSubscription	**EventSubscription** is the opposite of **EventPublication**. Any class that wants to subscribe to an event with a specific event URI needs to implement an event handler that matches the method signature of the **EventPublication**. You must mark this event handler with the [**EventSubscription**] attribute and the publisher's event URI (in the attribute's constructor) and CAB makes sure that a subscriber will be notified each time a publisher raises an event with a matching event URI and method signature.
Model	Model is the (client-side) business entity a WorkItem processes; for example, Customer, BankAccount, or Loan.
View	View is an ordinary .NET Framework user control that is responsible for presenting a part of or the whole model to the user and allowing the user to modify its contents through user interface controls. Typically, the view implements only UI logic; whereas the related client-business logic is implemented in the presenter/controller.
SmartPart	A SmartPart is a .NET Framework user control with the [**SmartPart**] attribute applied. Optionally, a SmartPart can implement the **ISmartPartInfoProvider** interface. As such, it provides additional information about itself, such as a caption or a description.
Controller	Controller is a class that implements the logic for modifying a model. One controller can modify a model presented by many views. Origin: Model-View-Controller
ObjectBuilder	ObjectBuilder is a foundation component of CAB that acts as the factory for creating objects that require specific builder strategies to be created or that need features, such as automatic instantiation and initialization of dependant objects when creating instances. As such, the ObjectBuilder has a combined role as a factory, a dependency injection framework, and a builder strategy framework.
Dependency Injection	Dependency Injection is a pattern that allows a factory to automatically create or initialize properties or members of objects with dependant objects. ObjectBuilder provides this functionality.

Understanding these terms and their roles in CAB is crucial for any good composite smart client design that is based on CAB. Nevertheless, this white paper does not focus on the base classes and APIs of CAB; therefore, for more information, take a look at the CAB documentation or the MSDN articles referenced in the section "Scope of This Whitepaper" earlier in this white paper.

Smart Client Software Factory

Although CAB provides a great infrastructure, getting started with CAB is definitely a challenge. Furthermore, working with CAB requires developers to complete many manual steps, such as creating classes inherited from the WorkItem base class to create a use case controller or manually creating Controller classes, View classes, and Model classes. The Smart Client Software Factory is an extension to Visual Studio 2005 Professional (or a later version) that provides added functions for automating these tasks and provides huge, detailed documentation about creating composite smart clients using CAB, including how-to topics and a fantastic set of reference implementations, such as the Global Bank Reference Implementation.

The automation of developer tasks of Smart Client Software Factory (SCSF) is based on the Guidance Automation Extensions, which is also provided by the Microsoft patterns & practices team. The Guidance Automation Extensions are an infrastructure that allows architects and developers to easily create extensions into Visual Studio for automating typical development tasks with the primary goal of enforcing and ensuring common directives and guidelines for their projects. SCSF provides such guidelines for CAB-based smart clients and automates things like creation of new modules and use case controllers in addition to event publications and subscriptions. Of course, these guidance packages are completely extensible. This white paper assumes that you are familiar with the basic concepts of SCSF and Guidance Automation Extensions. But because there are a couple of terms that are especially important for understanding this paper, they are summarized in Table 2.

Table 2: Important Classes/Terms Required to Understand for SCSF

Class / Term	Description, Purpose
WorkItemController	A **WorkItemController** is a class introduced by the Smart Client Software Factory that encapsulates common initialization logic for a WorkItem. When creating WorkItems with SCSF, instead of directly inheriting from the WorkItem base class, you inherit from this class to get the added initialization logic.
ControlledWorkItem	Again, this is a class introduced by the Smart Client Software Factory; it is a generic class for instantiating new WorkItems based on a **WorkItemController**. It launches the added initialization entry points provided by the **WorkItemController** for you.
ModuleController	**ModuleController** classes are introduced by the Smart Client Software Factory and are used for encapsulating a special WorkItem within a module taking on the role of a root WorkItem within a module. The default **ModuleInit** implementation of a module created with the Smart Client Software Factory automatically creates a WorkItem for a module named **ModuleController**. Therefore, the **ModuleController** is the primary entry point for all WorkItems provided by a module.
Presenter	Presenter is a class that implements the logic for one single

	SmartPart (view). The presenter is based on the Model-View-Presenter (MVP) pattern, which is basically a simplified variant of the Model-View-Controller (MVC) pattern. The big difference between MVP and MVC is that with MVP, the View is completely controlled by the presenter, whereas in the MVC, the controller and the model can update the view.

For more information about SCSF and Guidance Automation Extensions, see the following resources on MSDN:

- "Guidance Automation Extensions and Guidance Automation Toolkit": http://msdn.microsoft.com/vstudio/teamsystem/workshop/gat/default.aspx
- "Introduction to the Guidance Automation Toolkit": http://msdn.microsoft.com/vstudio/teamsystem/workshop/gat/intro.aspx
- "Smart Client Software Factory": http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/scsflp.asp

Primarily, Smart Client Software Factory is a very useful tool that supports developers to help them create CAB-based smart clients with minor effects on architectural decisions—it dramatically increases developer productivity because of its integrated, automated developer tasks into Visual Studio 2005.

Outlining the Development Process

As mentioned earlier, developing smart clients based on the Composite UI Application Block is very use case–centric. Therefore, it is crucial to start having a good understanding of your use cases (in the context of the client application). Primarily, there are three aspects that affect the development process for CAB-based smart clients: the Shell, infrastructure services, and the actual, detailed use cases. Therefore, designing composite smart clients takes place in three iterations:

1. Start achieving a high-level understanding of requirements common to most of the use cases. Common UI-related requirements are candidates for being integrated directly into the shell or at least affect the shell's layout and design. Non-UI related functionality that is common to all (or most of) the use cases are candidates for central services.
2. Create detailed use case diagrams. Use cases are good candidates for becoming WorkItems (and sub-WorkItems) in your application design. Relationships between use cases are candidates for using either the Command pattern or the event broker subsystem of CAB. Logically, closely related WorkItems, such as WorkItems implementing use cases for the same actors (roles of users in your company), are good candidates for being packaged into modules together.
3. Refine the use case diagrams, and then analyze relationships between use cases and reusability and security aspects of your use cases (WorkItems). During this refinement, you might need to slightly refactor your WorkItem packaging. Typical findings are things such as WorkItems that are reused independently of others packaged into the same module or used isolated from their parent WorkItems so that they are better suited as first-level WorkItems. A first-level WorkItem doesn't have a parent WorkItem other than the **RootWorkItem** of the shell (or the **ModuleController** WorkItem introduced by the Smart Client Software Factory).

Figure 1 outlines the overall development process; therefore, it gives you a first impression of how CAB affects the development process.

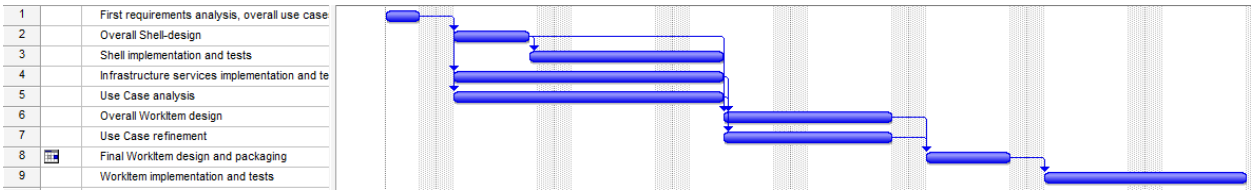


Figure 1: Development Process Outlined Schematically

The first requirements analysis needs to be done by the core team. Based on this first requirements analysis, the core team can start designing the shell and infrastructure services. First and foremost, it is essential that the team identifies the core requirements to these infrastructure services and the shell. These requirements need to be expressed in service interfaces (services are registered based on types in CAB that should be (but do not need to be) expressed through interfaces).

At the same time, a second team, together with the business departments, can start with a detailed use case analysis. In the meantime, members of the core team can start developing the shell and the necessary infrastructure services. The shell and infrastructure services are implemented one time at the

very beginning. When finished implementing these services, all the development effort takes place in developing WorkItems that are implementing the actual client side parts of the business requirements. Therefore, the shell and infrastructure services are typically implemented only once.

Because they are used by nearly every WorkItem, it is important that both the shell and the infrastructure services are nearly complete before you start developing a broad range of WorkItems. Furthermore, before you start designing and developing masses of WorkItems, you should start with a small number of core WorkItems to verify the design and implementation of the shell. Maybe you will decide to add some minor new requirements on the shell and the infrastructure services that require you to change the service interfaces (of course, building a prototype before starting the actual project probably makes sense in your case as well—but that's your decision).

The use cases identified during the detailed use case analysis are the foundation for identifying WorkItems in CAB. When taking the use case–driven approach for designing WorkItems, each use case maps to a single WorkItem. Relationships between use cases in your use case diagram are indicators for sub-WorkItems, communication between WorkItems through events or commands. Typically, a refinement and a detailed analysis of the relationships between use cases are indicators of which type of relationship there is (parent-child, event, or command). This refinement will also affect your strategy for packaging WorkItems. Therefore, it is essential to keep this refinement iteration in mind before you start developing the broad range of WorkItems. You will get more details about each of these steps in the following sections of this white paper.

Finally, the overall design of a composite smart client based on CAB will adhere to a layered approach where CAB provides the fundamental framework; the shell and the infrastructure libraries are your application foundation (maybe a framework based on top of CAB) and the modules containing the WorkItems are the actual business applications dynamically plugged into the shell. So CAB can be seen in two different ways: as a ready-to-use infrastructure for building pluggable smart client applications and as a meta-framework that allows you to build your own business application framework on top of CAB, which is composed by your shell and infrastructure services as shown in Figure 2.

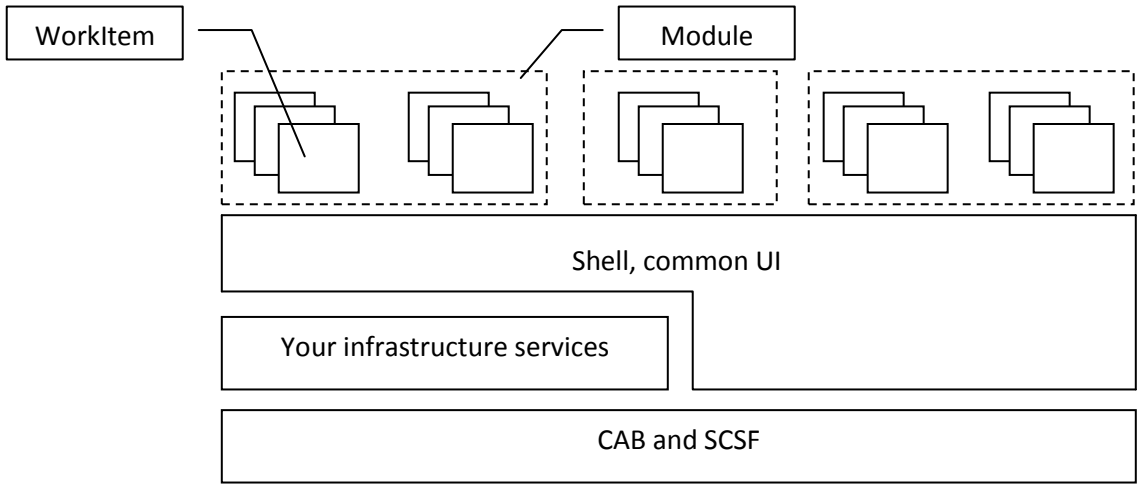


Figure 2: Layering of CAB-based Smart Clients

Application Shell-Service and Infrastructure Service Design

One of the first parts you will design and implement for your CAB-based smart client is the shell. The shell is the actual application that is responsible for loading and initializing base client services, loading and initializing modules, and providing the basic UI of the application and hosting the SmartParts loaded by first-level WorkItems (remember, first-level WorkItems don't have a parent other than the **RootWorkItem** or a **ModuleController**). The basic UI of an application is equal for each and every module loaded into the application. As such, it must fulfill requirements common to all use cases. Typical examples for common user interface elements that need to be added to the shell are the following:

- Menus, toolbars, and tool strips
- Navigation controls, such as the Outlook toolbar you are familiar with from Outlook 2003/2007
- A common message pane for having a central point to display messages for the user (such as errors or warnings)
- Task panes, such as the ones you know from Office 2003/2007, or taskbars, such as the one you know from the Windows XP Explorer
- Context panes that display information based on the current situation of the user

These are just some examples for good candidates to be integrated into the shell. With workspaces and UIExtensionSites, CAB provides a ready-to-use infrastructure for such extensibility points in the shell. Workspaces are used for completely replacing parts of the UI, whereas a UIExtensionSite is used for extending existing parts of the shell, such as adding menu entries or adding tool strip controls. Figure 3 shows a typical example for a shell.

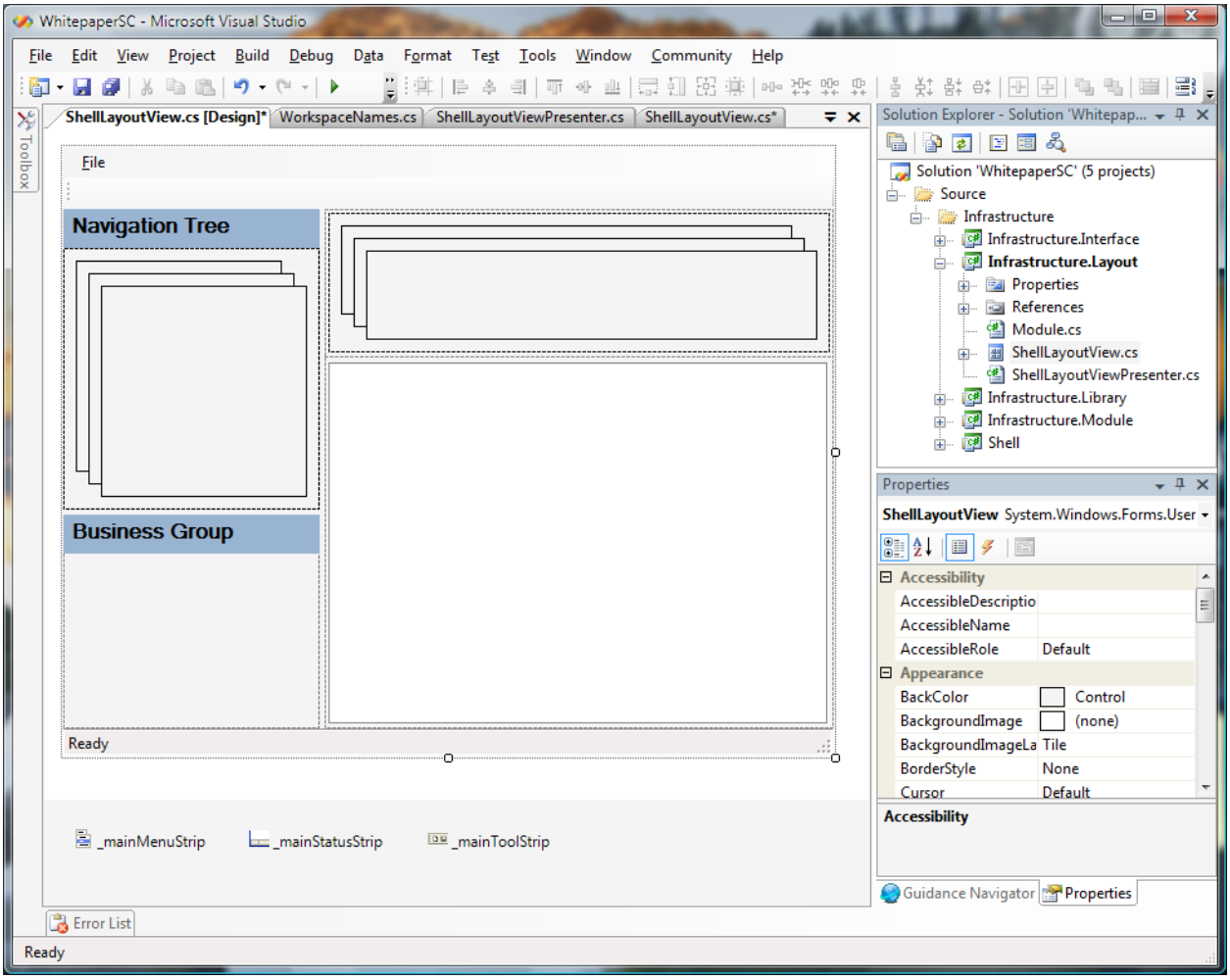


Figure 3: A Typical Example for a Shell with Workspaces and UIExtensionSites

Workspaces and UIExtensionSites as you see them in Figure 3 are publicly available to all services and modules loaded into the smart client application. CAB allows you to access these in a very generic fashion as follows:

```
workItem.UIExtensionSites["SiteName"].Add<ToolStripButton>(
                                        new ToolStripButton());
```

Although this approach gives you maximum flexibility, you might want your developers to have a "strongly typed" way to access the shell. Furthermore, this introduces a very tight coupling to the contents of the shell. Therefore, it is recommended to have a layer of indirection between the shell and components that you want to add to the shell. Based on the functionality, the shell needs to provide to other components; you can design an interface implemented by the shell (or the shell's main view presenter) for extending the shell and register the shell as a central service used by other components of CAB, as shown in Figure 4.

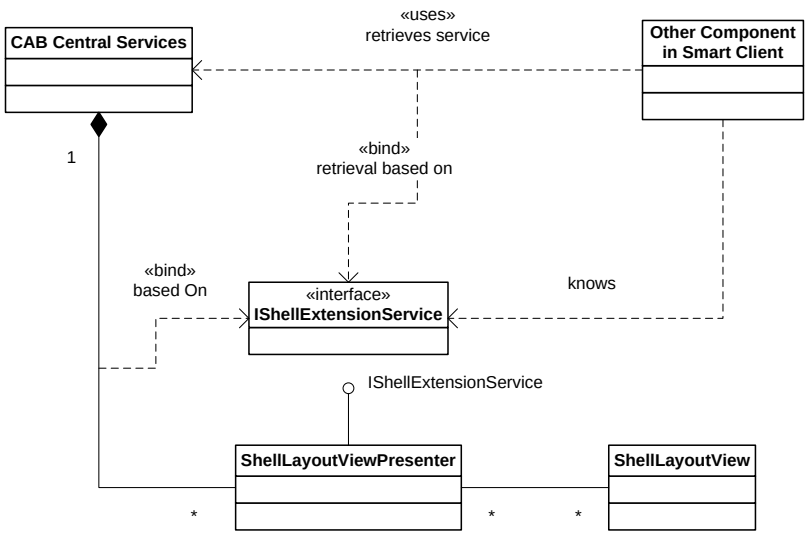


Figure 4: Shell Implementing Your Custom IShellExtension Interface Provided as a Central Service

This is a very simple yet powerful concept because the components loaded into the smart client (either modules with WorkItems or modules with central services) do not need to know details about workspace names and UIExtensionSite names. Components loaded into the smart client can access these services as follows:

```
IShellExtensionService ShellService =
    WorkItem.Services.Get<IShellExtensionService>();
ShellService.AddNavigationExtension(
    "Some new Menu", null, someCommand);
```

The implementation of this **IShellExtension** interface can still leverage this infrastructure to keep the shell UI design decoupled from the shell service implementation as follows:

```
public class ShellLayoutViewPresenter
    : Presenter<ShellLayoutView>, IShellExtensionService
{
    // ...
    // Other code goes here
    // ...

    public void AddNavigationExtension(...)
    {
        // Create a new ToolStripButton
        ToolStripButton NewButton = new ToolStripButton();
        // ...

        WorkItem.UIExtensionSites[
               UIExtensionSiteNames.OutlookNavBar
        ].Add<ToolStripButton>(NewButton);

        // ...
    }

    public void ShowInWorkspace(...)
    {
        // ...
        WorkItem.Workspaces[
```

```
            WorkspaceNames.ContextWorkspace
        ].Show(smartPart, info);
        // ...
    }

    // ...
}
```

It is important to understand that the shell interface exposes only shell UI–related functionality as a service to other components. That means it is the right point of access to allow components loaded into the smart client to extend menus, tool strips, a taskbar displayed in the left part of the window, or add a message to a common message pane (as introduced in the examples earlier in this section).

Workspace vs. UIExtensionSite

As mentioned earlier, workspaces are used to replace a complete part of the shell with a different UI. Typically, this is done by WorkItems loaded within a module into your application. UIExtensionSites are static parts of the shell that may not be replaced at all. But UIExtensionSites can be extended by modules loaded into the smart client. Typically, they are used for launching a WorkItem (use case); therefore, in most cases they are added by **ModuleInit** components.

Direct Integration in the Shell vs. Encapsulation into a Separate Module

When it comes to common parts of the user interface, you typically need to decide if you want to add them directly into the main view of the shell or add them into separate infrastructure modules. The primary question here is *reusability*. Do you want to reuse the common UI part elsewhere? Maybe use it in other smart client shells? Or maybe use it in other modules that are displayed as a part of a view of a WorkItem? If yes, then of course you need to package it into a separate module instead of directly integrating it into the shell.

The second factor for this decision is configuration and security. If you need to dynamically load the UI part, or if you need to load it based on the end user's security context, you should also put this common UI into a separate module.

Putting the Shell's Layout in a Separate Module or Directly in the Application

Of course, you can also put the shell's main view with the overall layout into a separate module. Actually, the Smart Client Software Factory asks you this question when creating a new project. But how do you make a decision about this? Well, again it's fairly easy: if you want to dynamically exchange the shell's layout based on the security configuration or the common configuration of your smart client, you have to put the shell's layout into a separate module. If not, putting the shell's layout into a separate module is an unnecessary overhead.

Encapsulate UI in User Controls

At this point, you may be wondering what happens if you decide to add the UI directly to the shell or add the layout directly to the main application instead of putting it into a separate module because requirements are now changing and you need to outsource these components into separate modules.

Well, to keep your refactoring overhead minimal, in any case, you should encapsulate complex parts of the UI into separate user controls and keep the logic of the UI separated into a presenter class. With these patterns in mind, refactoring the application later is not a big deal anymore.

Infrastructure Services

As mentioned earlier, infrastructure services encapsulate functionality common to all (or common to many) modules and components loaded into the smart client. Unlike shell infrastructure, these services are not bound to UI-specific tasks. They encapsulate client-side logic (maybe client-side business logic for offline capability). "Out-of-the-box," CAB introduces many infrastructure services, such as an authentication service or a service for loading a catalog of modules configured somewhere (by default, in the ProfileCatalog.xml file stored in your application directory). Of course, you can also introduce your own services. Typical examples for central infrastructure services not introduced by CAB are the following:

- Authorization service ("out-of-the-box," CAB and SCSF do not provide one, but the **BankBranchWorkbench**, which is one of the reference implementations provided by SCSF, demonstrates how to use the **ActionCatalog** provided by SCSF to implement one)
- Web service agents and proxies encapsulating calls to Web services and offline capability
- Context services to manage user context across WorkItems
- Configuration services for accessing application-centric configuration
- Deployment services using ClickOnce behind the scenes for programmatic, automatic updates

Always start with defining interfaces for your common services because CAB allows you to register central services based on types as follows:

```
MessageDisplayService svc = new MessageDisplayService(_rootWorkItem);
_rootWorkItem.Services.Add<IMessageDisplayService>(svc);
```

Infrastructure services need to be encapsulated into separate infrastructure modules and loaded before other modules with actual use case implementations are loaded.

Identifying WorkItems

WorkItems are components responsible for encapsulating all the logic for specific tasks the user wants to complete with the application. As such, WorkItems are the central and most important parts of your composite smart client because they provide the actual business functionality to the users. For this purpose, a WorkItem itself contains or knows about one or more SmartParts (views) with their controller classes and models. The WorkItem knows which SmartParts need to be displayed at which time and which sub-WorkItems need to be launched at which time and it manages state required across SmartParts and sub-WorkItems. Each CAB application has one **RootWorkItem** that acts as the central entry point for global services and WorkItems added by modules. With SCSF, every module automatically loads a **ModuleController** WorkItem, which acts as a root WorkItem within a module. For the remainder of this paper, WorkItems, which are directly added to either the **RootWorkItem** or the

ModuleController without having any other parents in between, are referred to as first-level WorkItems. These are the entry points into a specific business-related task.

Basically, there are two ways for identifying WorkItems for your composite smart client: a use case-driven approach and a business entity–driven approach.

Use Case-Driven Strategy

One of the biggest advantages of the architecture of CAB is that it allows you to design your WorkItems based on use case diagrams. Actually, in most cases, you will have a 1:1 mapping between use cases and WorkItems—typically, one use case will be encapsulated into a WorkItem. Therefore, WorkItems are nothing else than use case controllers implementing the UI processes necessary for completing a use case (task) and putting together all the required parts for doing so. Take a look at the use case diagram illustrated in Figure 5. This use case diagram was designed for a stock management system, which is used for consulting customers for stock operations and fulfilling customer stock requests.

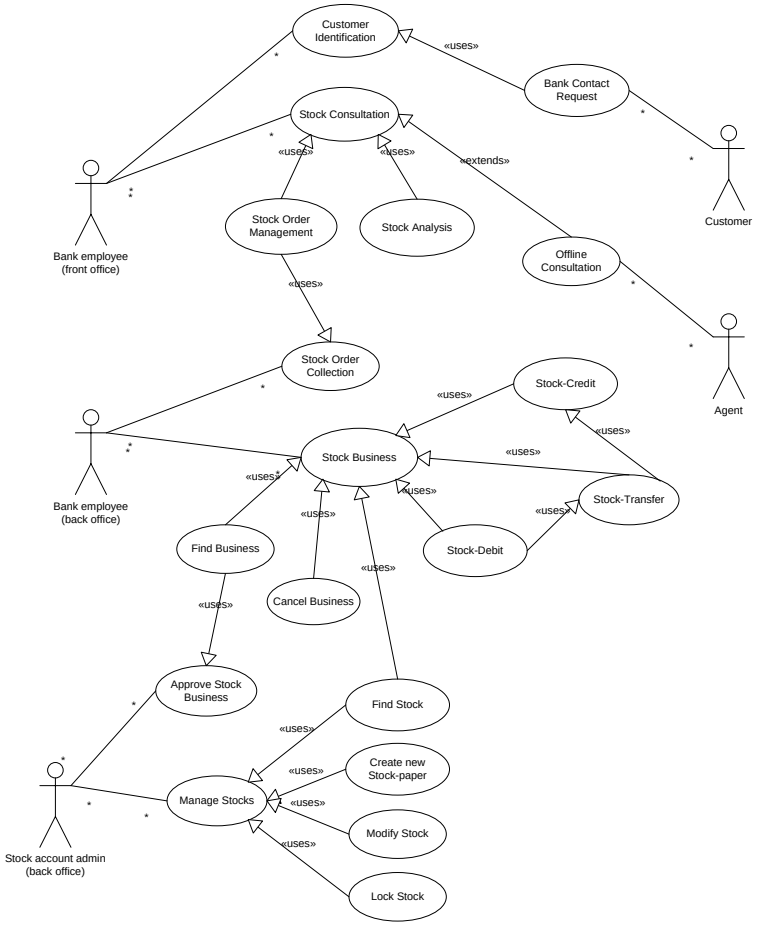


Figure 5: A Sample Use Case Diagram

Suppose you need to create a smart client application used by bank employees. The application supports front office employees for consulting customers and supports back office employees for completing the stock business transactions. First, you map one use case to one CAB WorkItem.

Relationships between use cases can be of two different types: a use case is either a sub-use case of another one or a use case is used by many other use cases and not only by its own parent. Of course, a use case that is really just a sub-use case and is not reused by others results in a sub-WorkItem. Pure sub-use cases are sub-WorkItems that are not accessible from outside their parents; use cases used by many other use cases in addition to its own parent need to be accessible either directly or through their parent WorkItem.

The Role of Root Use Cases

Root use cases—resulting in first-level WorkItems (or if there is just one, they can be directly implemented in the **ModuleController**)—are the entry points for all their sub-WorkItems. They are responsible for managing state required for all sub-WorkItems and providing the right context for all their sub-WorkItems. It is recommended that first-level WorkItems accomplish the following steps when being loaded:

1. Add services required by this set of WorkItems.
2. Retrieve service references required by this WorkItem.
3. Register commands for launching sub-WorkItems.
4. Add UIExtensionSites to allow starting into WorkItem-code via previously registered commands.
5. Create instances of sub-WorkItems when they are needed.
6. Create and register instances of SmartParts required by the WorkItem itself.

Typically, simple WorkItems without sub-WorkItems or sub-WorkItems themselves execute the same steps, except that they should not register services and add UI extension sites. Furthermore, sub-WorkItems should not register commands as they are called through their parents.

First-level WorkItems themselves are created by the **ModuleInit** of the module they are contained in. A **ModuleInit** of a module registers commands and UIExtensionSites for starting functionality encapsulated in the first-level WorkItem, but only if it should not be automatically createdand started.

Exclusion of Use Cases

When taking a closer look at the use case diagram, you will see that not all use cases will result in WorkItems. Take a look at the "Bank Contact Request" use case. Remember that for now, you want to design a smart client used only by bank employees. A customer requesting a stock consultation (expressed through the "Bank Contact Request") can either go directly to the front desk or use an Internet banking solution or something else. In any case, this use case is nothing that needs to be integrated into the smart client for the bank employees. Instead, "Customer Identification" needs to be part of the smart client you are now designing.

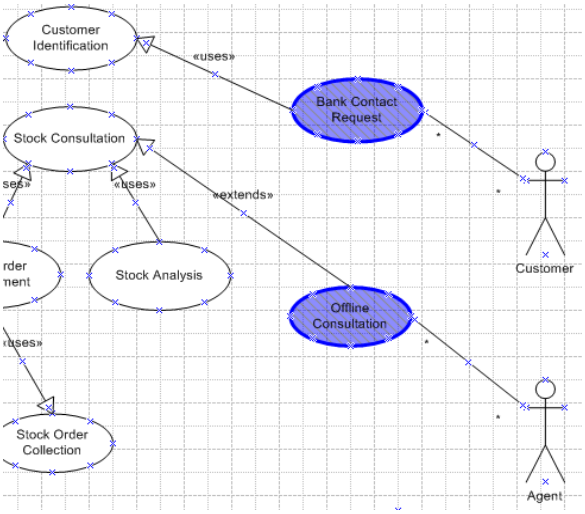


Figure 6: Exclusing Use Cases

Another example is the "Offline Consultation" use case. What does it mean to your smart client? Is it really going to be a separate WorkItem? Does it change anything in the business logic? No. Therefore, it won't be a separate WorkItem. But this use case is an indicator for something completely different; it's an indicator for the need of offline support. Therefore, this is something you need to verify against the infrastructure services identified earlier; for example, Web service agents need to support connection detection, offline reference data stores, and update message queues.

Sub-WorkItem or WorkItem

Some WorkItems will become first-level WorkItems although they appear just as sub-WorkItems in the use case diagram. Typically, this happens when a use case logically belongs to a parent use case; but in a detailed analysis, you see that it does not share state, context, or anything else with other sub-use cases and that it does not also depend on shared state, context, or other commonalities. To avoid unnecessary overhead in that case, you should also make these WorkItems first-level WorkItems. Take a close look at the use cases "Find Stock" and "Find Business" in the use case diagram of Figure 5. These use cases are used for finding stock papers or ordered business transactions. They do not depend on shared state or on context of the first-level WorkItem. Because of this, such WorkItems are perfect candidates for becoming first-level WorkItems themselves. Then they are being called through commands that are registered by the module of which they are part.

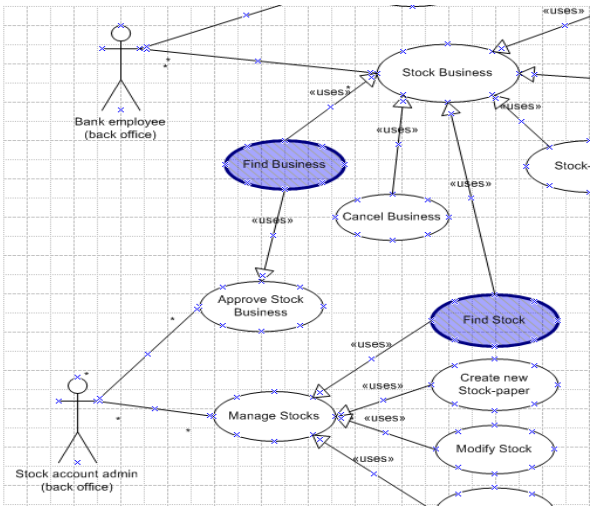


Figure 7: Root-WorkItem or Sub-WorkItem

WorkItem-Identification – Summary

To summarize, the following steps identify WorkItems based on use case diagrams for CAB-based composite smart clients:

1. Create the use case diagram.
2. Map each use case to one WorkItem in CAB.
3. Exclude use cases not directly related to your smart client.
4. Analyze relationships between use cases. If use cases are used by other use cases in addition to their parent, they are candidates for becoming first-level WorkItems.
5. Analyze requirements of use cases. If use cases do not depend on state or context of their parents, and vice-versa, they are also candidates for first-level WorkItems.

In the preceding list, steps 4 and 5 are parts of a use case refinement process where you analyze your use case diagrams again to identify the characteristics of relationships between use cases. Finally, Figure 8 shows the final class diagram that results from the use cases illustrated in Figure 5.

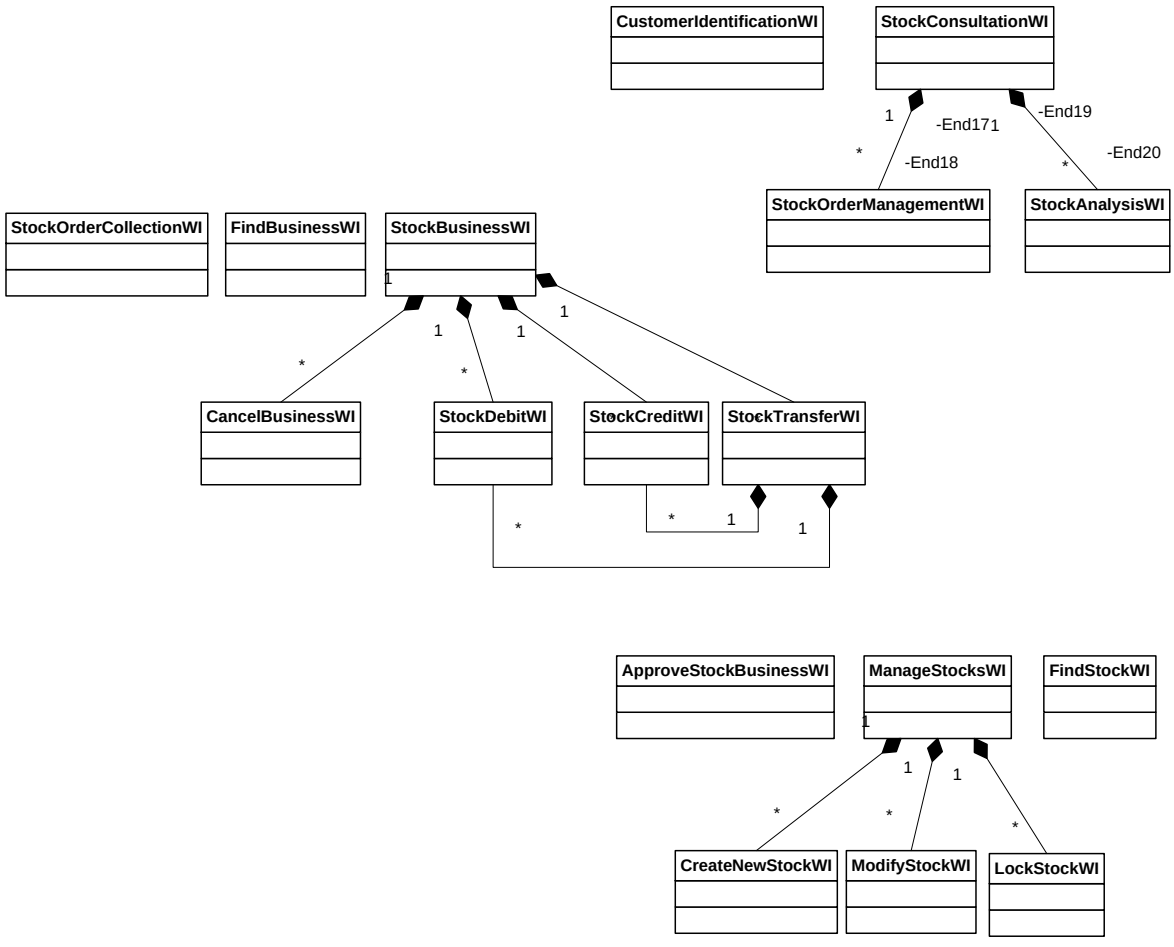


Figure 8: Class Diagram That Shows the WorkItems for the Previous Use Cases

Business Entity–Driven Strategy

A much simpler approach for smaller, simple applications is identifying and structuring WorkItems based on the business entities processed by the smart client application. You create a list of business entities processed by your application. Typical examples of business entities are Customer, Product, Stock, StockCredit, StockDebit, or StockTransfer. For each of these entities, you create a WorkItem. Because StockTransfer is a combination of StockCredit and StockDebit, it uses them as sub-WorkItems.

Packaging WorkItems into Modules

After you identify your WorkItems, you need to package them into modules. A module is a unit of deployment for CAB-based smart clients. Basically, you package logically related WorkItems that address the same space of business into a module. Using the use case diagram in Figure 8 as an example, you would create a module for stock consultation, one for stock business WorkItems, and a last one for stock management, as shown in Figure 9.

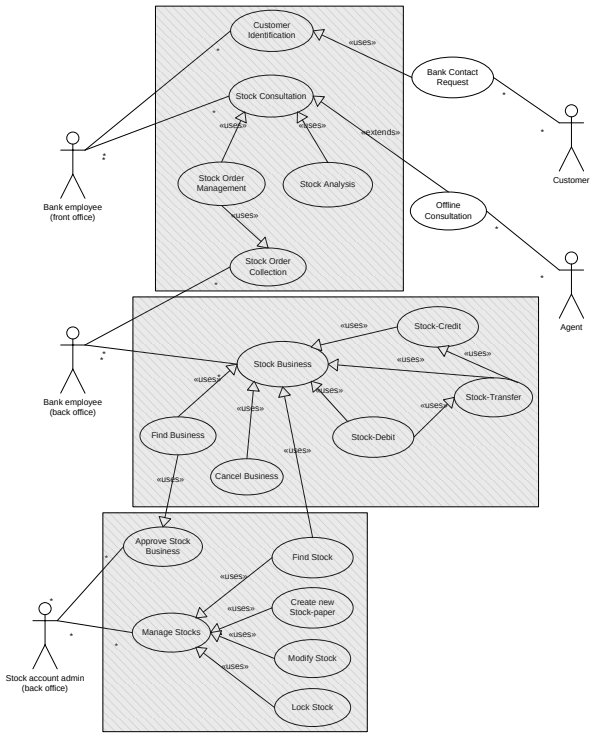


Figure 9: Packaging WorkItems into Modules

The following are also required for deciding how to package of WorkItems into modules:

- Security
- Configurability
- Reusability

First and foremost, modules are configured in a profile catalog. By default, this catalog will be retrieved from the ProfileCatalog.xml file in the application directory. These modules can be configured based on role membership of a user. Therefore, security plays a central role for deciding how to package

WorkItems into modules. Suppose a user is not allowed to manage any stocks, as shown in the preceding use cases. But users who are not allowed to create new stocks or to lock stocks will need to find stocks while completing their tasks. When configuring your stock management module (which contains the "Find Stock" WorkItem, according to the packaging illustrated in Figure 9) in a way that bank employees from the back office cannot use, they are also not allowed to use the "Find Stock" WorkItem. But because they also need this WorkItem, it is useful to package it into a separate module. Configurability and reusability are very similar reasons compared to security. If you want to reuse WorkItems independently from others, it makes sense to encapsulate them into separate modules. If you need to configure WorkItems independently from others, you also put them into separate modules.

In the example in Figure 9, this is true for "Find Stock," "Find Business," and "Stock Order Collection" WorkItems. Therefore, it is useful to encapsulate them into separate modules, as shown in Figure 10.

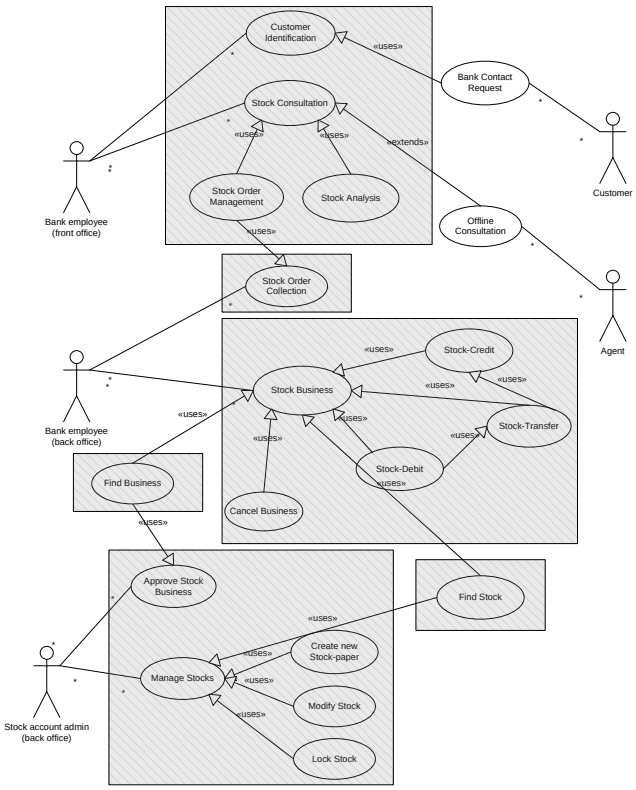


Figure 10: New Packaging According to Security and Reusability Requirements

If you decide that configuration requirements, security requirements, and reusability requirements are equal (or nearly the same) for some of the WorkItems that are outsourced into separate modules according to Figure 10, you also can package them into one module instead of three modules. For example, if the security requirements, configuration requirements, and reusability requirements of all "Find *X*" WorkItems are equal, you can create one module that contains all the "Find *X*" WorkItems.

CAB Infrastructure Usage Guidelines

Relationships between use cases represent interactions between use cases. In general, CAB supports two types of interactions between components—active communication through an implementation of the command pattern and passive communication through its event broker system. But when should you use each of them?

Using the Event Broker System

The CAB event broker provides a pre-built infrastructure for loosely coupled events. That means any component loaded into CAB can publish events, and other components can subscribe to these events without knowing who the publishers are. The relationship between publishers and subscribers is built through event URIs. Event URIs are strings that uniquely identify an event.

Therefore, use the event broker for loosely coupled types of communication. That means if a CAB component (WorkItem, controller, presenter, SmartPart, service) wants to provide information to others without getting immediate response and without requiring knowing who receives events, the event broker is the right way to go. If you need immediate response or need to know exactly who a component is talking to, the event broker is the wrong system. Do not implement immediate response by raising events back and forward between components. Keep relationships between components as events at a minimum—as few as possible and as much as necessary.

Using the Command Pattern

If your component needs something to be done immediately by another component or is passing control on to this other component, the Command pattern infrastructure can help accomplish these goals. Use commands for launching WorkItems (or sub-WorkItems) instead of events. Use commands to cause these WorkItems to executing functionality..

Part II: Developer's Guidance

The first part of this white paper focused on architectural guidance; this second part outlines the development process of a CAB-based smart client—from the creation of the shell to the creation of modules with WorkItems in the form of a step-by-step guide. Therefore, it can be seen as an extension to the guidance provided by CAB and the Smart Client Software Factory.

Developer's View on the Development Process

First, this white paper outlines the necessary development steps for creating a CAB-based smart client. This part of the white paper assumes that you have installed all the requirements mentioned in the "Requirements for Reading This White Paper" section of "Part I: Architectural Guidance for Composite Smart Clients." It will use the Smart Client Software Factory to complete the described tasks. The steps for developing a CAB-based smart client are outlined in Checklist 1.

Checklist 1: Steps for Creating a CAB-based Smart Client

1. Create a smart client project using SCSF.
2. Create the shell and its components, and design the shell's UI.
3. (Optional) Create and register services.
4. (Optional) Configure CAB infrastructure services if you have created ones in step 3.
5. Create CAB modules containing WorkItems with SmartParts and their presenters.
6. Configure and run the application.

Details on each of these steps are outlined in this part of the white paper. Each step is supported by Smart Client Software Factory to a certain extent. When using CAB without using the Smart Client Software Factory, you have to manually perform these steps.

Create a New CAB/SCSF Project

First, you need to create a new smart client project using the Smart Client Software Factory. For this purpose, complete the steps in Checklist 2.

Checklist 2: Creating a CAB/SCSF Project

1. Open Visual Studio 2005, select *File – New – Project* and select *Smart Client Application* from the *Guidance Packages – Smart Client* category.
2. In the first step of the wizard, you need to specify four things:
   - Location of compiled CAB assemblies. Note that you need to manually compile these after you install CAB.
   - Location of Enterprise Library assemblies. They will be automatically compiled when installing Enterprise Library.
   - The root namespace for your smart client project. This namespace is automatically used by SCSF for all modules added to the solution.

- If you want to add the layout of your shell into a separate module. This needs to be done when you want to also configure the shell's layout through the configuration and you want to be able to change the shell's layout through configuration.

3. Click **Finish**. SCSF creates a number of projects for you.

The Smart Client Software Factory adds a number of basic services commonly used for CAB-based smart clients. All these classes are based on the base classes provided by CAB, so they are consistent with anything CAB is offering. These basic services are encapsulated in different projects created by the SCSF. Table 3 contains detailed descriptions of the projects created by the Smart Client Software Factory for you when completing the Smart Client Application Wizard.

Table 3: Projects Created by the Smart Client Software Factory

Project	Description
Infrastructure.Interface	Defines interfaces for basic services created by SCSF. The added services created by SCSF are listed in Table 4.
Infrastructure.Library	Contains the implementations of the additional services generated by SCSF for you.
Infrastructure.Module	An empty module where you can add your own infrastructure services. By default, this module consists of an empty module controller (which is a default root WorkItem for a module). In this module, you should only add services used by the whole smart client application that is typically always loaded with your application independently of other modules.
Infrastructure.Layout	A project SCSF creates if you selected that you want to put the shell's layout into a separate module. It contains a SmartPart and a presenter for this SmartPart for the shell layout.
Shell	The actual Windows-based application of the smart client that contains the configuration and the actual main form that hosts the shell layout.

As mentioned earlier, SCSF generates interfaces and basic implementations for a number of services in addition to the three default services provided by CAB. Table 4 lists those services.

Table 4: Services Added by Smart Client Software Factory When Adding a New Project

Service	Description
Entity Translator Base Class	Base classes for building translators of entities. These are used for complex smart clients where you typically need to implement a mapping from the smart client's object model to the (Web) service's object models for loose coupling between those two.
Basic SmartPart Info Classes	CAB allows you to create classes that implement **ISmartPartInfo**. You need to implement this interface in a class when you want to display additional information for a SmartPart when it is being added to a workspace. For example, if you want to display a caption in a tab created within a tabbed workspace, you need to pass an implementation of **ISmartPartInfo** to the workspace so it displays the information. SCSF creates some basic implementations of this interface that are frequently used.

ActionCatalog Service	Allows you to define specific methods of your classes as actions that are automatically called when the security context of your user allows the user using these actions as explained in the sections "**Error! Reference source not found.**" and "Developing CAB Business Modules."
DependentModuleLoader	By default, CAB comes with a simple profile catalog and module loader for dynamically loading modules based on the configuration of the ProfileCatalog.xml file stored in the application's directory. SCSF adds a new module loader that allows you to specify relationships between modules in the configuration. With the **DependantModuleLoader**, you can specify that one module depends on others to be loaded before it gets loaded itself. By default, the **DependantModuleLoader** is used in SCSF-based projects.
WebServiceCatalog Service	With this added service, you can retrieve the modules to be loaded at startup of your smart client from a Web service. All you need to do is implement the Web service back-end and then configure this service in the smart client's application configuration.

To always automatically start the shell, you can set the project "Shell" as your startup project. Configuring the Visual Studio Solution with the project "Shell" as your start-up project is more convenient because, by default, Visual Studio automatically takes the currently selected project as your startup project.

Creating the Shell and Its Components

The first step after creating the project is designing your shell and the basic components that belong to your shell according to the requirements you have specified earlier in the project life cycle. Checklist 3 outlines the processes for creating the shell and its components.

Checklist 3: Creating the Shell and Its Components

1. Design the basic UI of the shell and add CAB UI components.
2. Register UIExtensionSites for UI parts that are extended but not replaced by loaded modules.
3. Create interfaces for UI-related basic shell services.
4. Implement and register UI-related basic shell services.

Again, the steps outlined in Checklist 3 are covered in detail in the following sections of this section in this white paper.

Design the Layout of Your Shell

This is actually the easiest part of the process. You just need to design the user interface of the shell using the standard Visual Studio 2005 Windows Forms Designer. You can also use the designer to add workspaces of CAB.

Checklist 4: Designing the UI of the Shell

1. Open the ShellLayoutView if the shell layout is put into a separate module (Infrastructure.Layout) or open the ShellForm if the layout is not put into a separate module.

2. Delete the existing controls you don't need—except menus, status strips, or tool strips because they will be required by UIExtensionSites.
3. Delete the existing DeckWorkspace controls if you don't need them.
4. If you delete existing workspaces, also perform the following steps:
    a. Switch to *WorkspaceNames.cs* in the folder *Constants* of the *Infrastructure.Interface*.
    b. Delete the constant of the workspace you have deleted.

```
public class WorkspaceNames
{
    public const string LayoutWorkspace = "LayoutWorkspace";
    public const string ModalWindows = "ModalWindows";
    // public const string LeftWorkspace = "LeftWorkspace";
    public const string RightWorkspace = "RightWorkspace";
}
```

    c. Switch to the code-behind of your shell layout UI, and delete the code that belongs to the workspace you have just deleted in its constructor.

```
public ShellLayoutView()
{
    InitializeComponent();
    // _leftWorkspace.Name = WorkspaceNames.LeftWorkspace;
    _rightWorkspace.Name = WorkspaceNames.RightWorkspace;
}
```

5. (Optional) Add the CAB controls to your Visual Studio 2005 controls toolbox:
    a. Open the toolbox, right-click it, and then select *Choose Items* from the context menu.
    b. In the **Choose Toolbox Items** dialog box, browse to the Microsoft.Practices.CompositeUI.WinForms.dll assembly.
    c. Now, in the **Choose Toolbox Items** dialog box, filter the items by the namespace **Microsoft.Practices.CompositeUI**, and check all items appearing in the dialog box, as shown in Figure 11.
6. Now you can drag-and-drop CAB controls onto your surface as you need them.

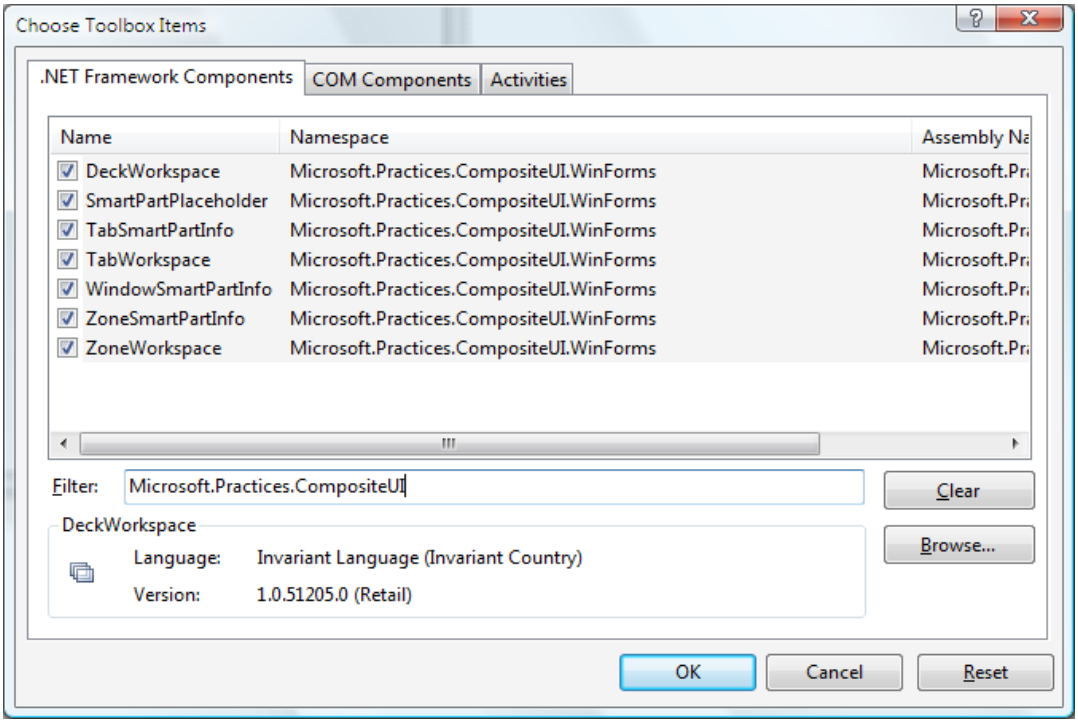


Figure 11: Choose Toolbox Items dialog box

For consistency when adding a workspace to your shell UI, you always need to complete the same set of steps as summarized in Checklist 5.

Checklist 5: Adding a Workspace to Your Shell's UI

1. Drag-and-drop one of the workspace controls from the toolbox onto your design surface.
2. Specify a name for the (Name) property of the workspace.
3. Switch to *Infrastructure.Interface – Constants – WorkspaceNames.cs* and add a string constant that contains the name for your workspace.

```
public class WorkspaceNames
{
    public const string LayoutWorkspace = "LayoutWorkspace";
    public const string ModalWindows = "ModalWindows";

    public const string NavigationWorkspace =
                                    "ShellNavigationWorkspace";
}
```

4. Rebuild the solution to update IntelliSense as weak references are used.
5. Open the code-behind of your shell UI control/form and assign the workspace name to the newly created workspace in its constructor as follows.

```
public partial class ShellLayoutView : UserControl
{
    private ShellLayoutViewPresenter _presenter;

    /// <summary>
    /// Initializes a new instance of the ShellLayoutView class.
```

```
        /// </summary>
        public ShellLayoutView()
        {
            InitializeComponent();
            //_leftWorkspace.Name = WorkspaceNames.LeftWorkspace;
            //_rightWorkspace.Name = WorkspaceNames.RightWorkspace;

            NavigationWorkspace.Name = WorkspaceNames.NavigationWorkspace;
        }

        // ...
        // More code goes here
        // ...
    }
```

Finally, that's it with the overall UI design. Next, you need to design UIExtensionSites and register them.

Register UIExtensionSites

Unlike workspaces, UIExtensionSites are fixed parts of the shell that cannot be completely replaced but can be extended by modules loaded into the application. Typical examples are menus, tool strips, or status strips of the application that are basically defined by the shell but can be dynamically extended as modules are loaded into the application. The basic steps for registering a UIExtensionSite are listed in Checklist 6.

Checklist 6: Adding a New UIExtensionSite

1. Add a menu, tool strip, tool strip item, status strip, or status strip item that you want to be extended by modules loaded into the application.
2. Add a constant for the UIExtensionSite to the *Constants.cs* file of the *Infrastructure.Interface*.

```
/// <summary>
/// Constants for UI extension site names.
/// </summary>
public class UIExtensionSiteNames
{
    /// <summary>
    /// The extension site for the main menu.
    /// </summary>
    public const string MainMenu = "MainMenu";

    /// <summary>
    /// The extension site for the main toolbar.
    /// </summary>
    public const string MainToolbar = "MainToolbar";

    /// <summary>
    /// The extension site for the main status bar.
    /// </summary>
    public const string MainStatus = "MainStatus";

    /// <summary>
    /// Custom ToolStrip UIExtensionSite added to the shell
    /// </summary>
    public const string OutlookNavBar = "OutlookNavBarSite";
}
```

3. Again, compile the solution to update IntelliSense because SCSF uses weak references.

4. Add a property to your shell view (*ShellLayoutView.cs* or *Shell.cs*, depending on whether you have put the layout into a separate module) so that the presenter (or the ShellApplication) has access to the added control for the UIExtensionSite.

```
/// <summary>
/// Gets the outlook navigation tool strip.
/// </summary>
internal ToolStrip OutlookNavigationToolStrip
{
    get { return this.OutlookToolStrip; }
}
```

5. Register the UIExtensionSite with your application. There are different cases you need to consider when registering the site:
    - When you want to register a drop-down item's collection, you need to register the MenuItems or Items collections of the menu or the tool strip.
    - If you want items to be inserted immediately after a specific MenuItem or ToolStripItem, you register this item as a UIExtensionSite.
    - If you want items to be entered into only a tool strip, status strip, or menu strip, you just register this tool strip, status strip, or menu strip as a UIExtensionSite.
6. Registering a UIExtensionSite needs to be done in code. There are two different situations you need to consider when registering the site:
    - When creating the shell in a separate module, open ShellLayoutPresenter.cs in your Infrastructure.Layout module and register the UIExtensionSite in the OnViewSet method of the presenter.

```
public class ShellLayoutViewPresenter
                : Presenter<ShellLayoutView>
{
    protected override void OnViewSet()
    {
        // UIExtensionSites registered by default
        WorkItem.UIExtensionSites.RegisterSite(...);
        WorkItem.UIExtensionSites.RegisterSite(...);
        WorkItem.UIExtensionSites.RegisterSite(...);

        // Custom UIExtensionSite (Outlookbar)
        WorkItem.UIExtensionSites.RegisterSite(
            UIExtensionSiteNames.OutlookNavBar,
            View.OutlookNavigationToolStrip);
    }
```

    - When creating the shell directly in the application, open the *ShellApplication.cs* source file in the *Shell* project, and add the UIExtensionSite on the *AfterShellCreated* method.

```
class ShellApplication
        : SmartClientApplication<WorkItem, ShellForm>
{
    // ...
    // Other code goes here
    // ...

    protected override void AfterShellCreated()
```

```
        {
            base.AfterShellCreated();

            RootWorkItem.UIExtensionSites.RegisterSite(...);
            RootWorkItem.UIExtensionSites.RegisterSite(...);
            RootWorkItem.UIExtensionSites.RegisterSite(...);

            // Custom UIExtensionSite (Outlookbar)
            WorkItem.UIExtensionSites.RegisterSite(
                UIExtensionSiteNames.OutlookNavBar,
                this.Shell.OutlookNavigationToolStrip);
        }


        // ...
        // Other code goes here
        // ...
    }
```

After you register a UIExtensionSite, you can use it in modules loaded into the smart client application. When registering new UIExtensionSites, you will recognize that SCSF creates a number of default UIExtensionSites for you. Checklist 7 outlines the steps for removing a UIExtensionSite.

Checklist 7: Remove Existing UIExtensionSites

1. Remove the control from the shell's UI.
2. Remove the property for the removed control in the ShellLayoutView.cs or ShellForm.cs.
3. Remove the UIExtensionSite registration in either the *AfterShellCreated* method of the *ShellApplication.cs* file or, when the shell UI is encapsulated into a separate module, remove the registration from the *OnViewSet* method of the *ShellLayoutViewPresenter.cs* presenter.
4. Remove the constant in the *UIExtensionSiteNames.cs* of the *Infrastructure.Interface* project.

Next, you can start creating shell services used by modules and components loaded into the smart client. These are central, shell UI–related services common to all WorkItems implemented by the application.

Create Interfaces for UI-Related Shell Services

Shell services are related to the UI of the shell. The first and foremost reason for defining such services is to put an indirection between UI parts of the shell and actual WorkItems that customize the shell's UI, although you can access any UIExtensionSite and workspace in a generic fashion, as shown in the following code example.

```
workItem.UIExtensionSites[
    UIExtensionSiteNames.OutlookNavBar].Add<ToolStripButton>(
                                            new ToolStripButton());
workItem.Workspaces[WorkspaceNames.NavigationWorkspace].Show(yourControl);
```

Putting some level of indirection between this type of access and the WorkItems makes things more convenient and avoids runtime errors caused by referencing non-existent UIExtensionSites and workspaces. Furthermore, it allows you to add additional, shell-related functionality exposed as a central service.

Checklist 8: Defining Central, Shell UI–Related Interfaces

1. Summarize all shell-based, central UI services.
2. Add a new interface to the *Infrastructure.Interface* project. Create an interface for each related group of shell-based UI services, such as IShellHelpService, IShellExtensionService, or IShellMessageService. For simple scenarios, it is enough to create just one interface and define all the functionality in one interface.
3. Define the methods for the interface according to the requirements of step 1.

Basically, that's it. Now you can start implementing and registering your shell-based, central services as outlined in the following sections.

Implement and Register UI-Related Basic Shell Services

Next, you need to implement the shell-based services and register them with the CAB infrastructure so they are available for all subsequently loaded modules. There are three different choices you need to consider:

- The shell-based service is closely related and tightly coupled to the shell's layout. In that case, it can be implemented by the layout's presenter in *Infrastructure.Layout* or in the shell's main form in the *Shell* project (if the layout is not put into a separate module).
- The shell-based service will be implemented in a separate class because it is not tightly bound to the layout, but it will not be reused in other applications. Then it's not necessary to put it into a separate module.
- The shell-based service will also be reused in other applications. Then it is useful to encapsulate it into a separate module. But if you do this, you need to also establish a contract between the shell and the shell-based service.

Checklist 9 outlines the steps for creating a shell service tightly coupled to the shell UI. The preceding list briefly describes the creation of shell services; they are created as normal foundation services modules. For more information, see the section "Creating and Registering New Services" later in this white paper.

Checklist 9: Shell-based Service Tightly Coupled to the Shell's UI

1. Add a new interface to the *Infrastructure.Interfaces* project.
2. Complete the interface definition according to the requirements. Keep in mind that this is a service tightly coupled to the shell, so include functionality with this characteristic only.

```
public enum ShellWorkspaceType
{
    NavigationWorkspace,
    ContextWorkspace,
    DetailsWorkspace
}

public interface IShellExtensionService
{
    void ShowInWorkspace(Control smartPart,
                         ShellWorkspaceType workSpace);
    void ShowInWorkspace(Control smartPart,
                         ISmartPartInfo info,
                         ShellWorkspaceType workSpace);
```

```
    void AddNavigationExtension(string caption,
                                Image icon, Command command);
}
```

3. Implement the interface in the **ShellLayoutViewPresenter** class of the *Infrastructure.Layout* project or in the **ShellApplication** class of the *Shell* project (if the layout is not encapsulated in a separate module).

```
public class ShellLayoutViewPresenter
    : Presenter<ShellLayoutView>, IShellExtensionService
{
    // ...
    // Other code goes here
    // ...

    public void AddNavigationExtension(string caption,
                            System.Drawing.Image icon, Command command)
    {
        // ...
    }

    public void ShowInWorkspace(Control smartPart,
                                ISmartPartInfo info,
                                ShellWorkspaceType workSpace)
    {
        // ...
    }

    public void ShowInWorkspace(Control smartPart,
                                ShellWorkspaceType workSpace)
    {
        // ...
    }

    #endregion
}
```

4. Register the service either in the *OnViewSet* method of the **ShellLayoutViewPresenter** class or in the *AfterShellCreated* method of the **ShellApplication** class.

```
public class ShellLayoutViewPresenter
    : Presenter<ShellLayoutView>, IShellExtensionService
{
    protected override void OnViewSet()
    {
        // UIExtensionSites registered by default
        // ...

        // Register the shell-extension service
        WorkItem.Services.Add<IShellExtensionService>(this);
    }

    // ...
}
```

5. Now the service can be used in every CAB-based component by retrieving it, as shown in the following code example.

```
IShellExtensionService ShellService =
```

```
    WorkItem.Services.Get<IShellExtensionService>();
ShellService.AddNavigationExtension(
    "Some new Menu", null, someCommand);
```

Creating and Registering New Services

After you have created and registered the shell UI–related services, you can start creating and registering other services commonly used by components loaded into your smart client. Basically, the steps for doing so are outlined in Checklist 10.

Checklist 10: Adding New Modules That Contain General Services

1. Add a new *Foundation Module* to the solution (from the Smart Client Software Factory context menu in your solution).
2. Activate the following option for the new module, deactivate all other options in the wizard:
    a. Create an interface library for this module
3. Add a new interface to the created interface project *YourSelectedName.Interface*, such as the following example.

```
public interface IMessageDisplayService
{
    void ShowMessage(string message);
    void ShowMessage(string message, string title);

    // ...
    // More options
    // ...
}
```

4. Add a new class to the *Services* folder of the created module *YourSelectedName* that implements the interface created in step 3.

```
public class MessageDisplayService : IMessageDisplayService
{
    private WorkItem ThisWorkItem;
    private IShellExtensionService ThisShellService;

    public MessageDisplayService(WorkItem workItem)
    {
        ThisWorkItem = workItem;
        ThisShellService =
            workItem.Services.Get<IShellExtensionService>();
    }

    #region IMessageDisplayService Members

    public void ShowMessage(string message)
    {
        ThisShellService.ShowStatusMessage(message);
    }

    public void ShowMessage(string message, string title)
    {
        ThisShellService.ShowStatusMessage(message);
        MessageBox.Show(message, title);
    }

    #endregion
```

```
}
```

5. Instantiate and register the service in the *Module.cs* class created by SCSF by overriding the *AddServices* method.

```
public class Module : ModuleInit
{
    private WorkItem _rootWorkItem;

    [InjectionConstructor]
    public Module([ServiceDependency] WorkItem rootWorkItem)
    {
        _rootWorkItem = rootWorkItem;
    }

    public override void Load()
    {
        base.Load();
    }

    public override void AddServices()
    {
        base.AddServices();

        // Add the new service
        MessageDisplayService svc =
                new MessageDisplayService(_rootWorkItem);
        _rootWorkItem.Services.Add<IMessageDisplayService>(svc);
    }
}
```

Now every component loaded through a module into the smart client after this module has been created and can access the service as follows.

```
IMessageDisplayService MessageService;
MessageService = _rootWorkItem.Services.Get<IMessageDisplayService>();
MessageService.ShowMessage("Some Message");
```

Finally, that's it—that's how you can create and register your own services. Typical examples for such services are Web service agents (proxies), security services, or context services. If you have an existing set of Web service proxies and you have explicitly exposed the interfaces for these proxies, all you need to do is create a module as outlined in Checklist 10, but you can skip the creation of an interfaces library and you can skip the creation of a service. All you need to do in this case is register the existing Web service proxies with their interfaces in the *AddServices* method of the *Module* class of your newly added module. After you have created the shell, shell-related services, and all infrastructure services you need, you can start implementing the actual business logic encapsulated in CAB business modules.

Developing CAB Business Modules

Business modules are encapsulating-related WorkItems packaged according to the guidelines explained in the section "Packaging WorkItems into Modules." WorkItems themselves are responsible for launching sub-WorkItems and SmartParts as required for completing a use case. The steps for creating such modules are outlined in Checklist 11.

Checklist 11: Steps for Creating a CAB Business Module

1. Add the new business module using the Smart Client Software Factory.
2. Add new WorkItems to the created module. Each WorkItem encapsulates a use case.
3. (Optional) Create a new sub-WorkItem.
4. Add state required for a WorkItem and its sub-WorkItems.
5. Add the SmartParts required for a WorkItem.
6. Create commands for launching first-level WorkItems.
7. Create an ActionCatalog for creating UIExtensionSites.
8. Publish and receive events as required.

The details for the steps outlined in Checklist 11 are covered in detail in the subsequent sections of this section.

Adding the New Business Module

First, you need to create a new business module with the *Add New Business* module of the Smart Client Software Factory context menu of the Visual Studio solution. Again, you can create a separate module for the interfaces. Creating a separate interface library for your business module is important whenever you need to share types with other modules, including:

- Interfaces for services registered by this business module.
- Event arguments for events published by this business module.
- Constants for command names and event URIs provided and published by this business module.

Table 5 outlines the parts and their roles that are created when you add a new business module. These are part of the actual business module; the interfaces module just consists of the Constants folder; it does not include the other parts.

Table 5: Parts Created When Adding a New Business Module

Part	Description
Module class	Implementation of the **ModuleInit** base class from the Composite UI Application Block. This is for loading all the services and the first-level WorkItems of a module.
ModuleController	The SCSF does not directly create WorkItems; it creates **WorkItemControllers** that implement the actual use case logic. SCSF creates a root **WorkItemController** for every business module. This module-level root **WorkItemController** is the entry point for all sub-WorkItems (if there are any) and is called ModuleController.
Constants folder	Contains files for defining constants for UIExtensionSites, workspaces, command names, and event topics that belong to this business module.

After you have added the module, you can start creating WorkItems and SmartParts required for implementing the logic of the use cases implemented in the module.

Adding a New WorkItem

Unfortunately, SCSF does not include a separate guidance package for adding a new WorkItem. Therefore, you need to manually create the WorkItem as outlined in Checklist 12.

Checklist 12: Creating a New WorkItem

1. Create a new folder for anything that belongs to your WorkItem in the module's project.
2. Add a new class and add the following CAB-related namespaces and the namespaces that point to the infrastructure interfaces of your smart client (*Infrastructure.Interfaces* and namespaces of your other, required service interfaces). Typically, you need CAB namespaces for the Command pattern and utility classes. If you want to raise events, you also need the event broker namespace.

```
using System;
using System.Collections.Generic;

using Microsoft.Practices.CompositeUI;
using Microsoft.Practices.CompositeUI.Utility;
using Microsoft.Practices.CompositeUI.Commands;
using Microsoft.Practices.CompositeUI.EventBroker;

using CabWp.TestClient.Infrastructure;
using CabWp.TestClient.Infrastructure.Interface;

namespace CabWp.TestClient.Modules.BrowserModule.BrowserWorkItem
{
    public class BrowserWIController
    {
    }
}
```

3. The newly created class needs to be inherited from **WorkItemController**. If **WorkItemController** instances are reused, you should implement two methods: one for initializing the WorkItem for the first time and one for re-activating it if it is already running (typically called Run() and Activate()).

```
public class BrowserWIController : WorkItemController
{
    public void RunWorkItem()
    {
        // Retrieve a reference to IShellExtensionService

        // Create instances of SmartParts,
        // and sub-WorkItems and register them
    }

    public void ActivateWorkItem()
    {
        // Show instances of SmartParts in Workspaces
    }
}
```

4. Now you can create SmartParts and sub-WorkItems as outlined in the following sections.

As you can see in step 3 of Checklist 12, the methods for running and activating the WorkItem have now parameters defined—no workspaces or UIExtensionSites passed in. That means this is a first-level

WorkItem that requires the **IShellExtensionService** introduced in the section "Creating the Shell and Its Components" earlier in this white paper to acquire a reference and add SmartParts through this service. Sub-WorkItems need to get workspaces as parameters to the methods defined in step 3 and just use them for adding their SmartParts to specific workspaces of the parent WorkItem.

Create a New Sub-WorkItem

Basically, the steps for creating a sub-WorkItem are the same as outlined in Checklist 12, but the RunWorkItem() and ActiveWorkItem() methods are implemented slightly different. Instead of accessing workspaces of the shell directly through the **IShellExtensionService**, sub-WorkItems need to get workspaces they are working with from their parent WorkItem because these workspaces do not need to be part of the shell layout directly—they could be a part of a SmartPart of the parent WorkItem. In any case, sub-WorkItems should not know where workspaces they are working with are coming from to keep their reusability as high as possible.

```
public class SomeSubWorkItemController : WorkItemController
{
    private IWorkspace MyWS1, MyWS2...;

    public void RunWorkItem(IWorkspace ws1, IWorkspaces ws2...)
    {
        // Initialize the workspaces
        MyWS1 = ws1;
        MyWS2 = ws2;

        // Create instances of SmartParts,
        // and sub-WorkItems and register them
    }

    public void ActivateWorkItem()
    {
        // Show instances of SmartParts in Workspaces
    }
}
```

The rest is completely the same as the case for first-level WorkItems. Typically, sub-WorkItems are called only through first-level WorkItems.

Manage [State] in WorkItems

A WorkItem manages state for its SmartParts and sub-WorkItems. Typically, you keep the business entities (models) or parts of them in the state bag provided by the WorkItem base class of CAB. Checklist 13 outlines the necessary steps for adding state to a WorkItem.

Checklist 13: Adding State to a WorkItem.

1. (Optional) Create a class for the data stored in your state (this step is not necessary if you already have your business-entity = model defined or if the class is already provided by other parts, such as the base class library of the .NET Framework).
2. Add a public constant for a unique name of your state instance in the state bag of the WorkItem.

```
public class BrowserWIController : WorkItemController
{
    public const string BrowserHistoryStateName
```

```
                                = "BrowserHistoryState";

    // ...
    // Other code
    // ...
}
```

3. Add an instance of your state class or model to the state bag of the WorkItem. It is important that you add state before you create the SmartParts and sub-WorkItems for the case they need to access state already for initialization.

```
public class BrowserWIController : WorkItemController
{
    // ...

    public void RunWorkItem()
    {
        // Add state to the WorkItem
        WorkItem.State[BrowserHistoryStateName] =
                 new Dictionary<string, string>();

        // Create the sub-WorkItems and SmartParts
    }

    // ...
}
```

4. Add a property to your WorkItem for conveniently accessing the state content.

```
public Dictionary<string, string> BrowserHistory
{
    get
    {
        return WorkItem.State[BrowserHistoryStateName]
                          as Dictionary<string, string>;
    }
}
```

5. Now state can be used in sub-WorkItems or SmartParts as outlined in the following sections.

Checklist 14: Consuming State in CAB Components

- Option 1: Add a property to your sub-WorkItem, SmartPart, or presenter of a SmartPart with the state attribute applied. The name of the state must be the same as the one selected for initializing the state, as shown in Checklist 13. When using this option, the state must be initialized before ObjectBuilder creates the item.

```
private Dictionary<string, string> _BrowserHistory;

[State(BrowserWIController.BrowserHistoryStateName)]
public Dictionary<string, string> BrowserHistory
{
    get { return _BrowserHistory; }
    set { _BrowserHistory = value; }
}
```

- Option 2: You can directly access state through your WorkItem as follows:

```
Dictionary<string, string> hist;
```

```
hist = WorkItem.State[BrowserWIController.BrowserHistoryStateName]
                                        as Dictionary<string, string>;
```

Add SmartParts to WorkItems

Next, you can add SmartParts to your business module in the folder of the parent WorkItem for the SmartPart. Fortunately, SCSF includes a guidance package for this purpose, as described in Checklist 15.

Checklist 15: Add a New SmartPart with a Presenter

1. Right-click the folder created for the WorkItem of the previous selection.
2. From the context menu, select *Smart Client Factory – View (with Presenter)* as shown in Figure 12. SCSF creates three classes: a user control representing the view, an interface for decoupled communication between the presenter and the view, and a presenter implementing the logic for the view.
3. Design the UI of the user control using the standard Windows Forms designer.
4. Add method definitions to the interface created by SCSF for the view (*IViewName.cs*).

```
public interface IBrowserView
{
    void UpdateBrowser(string url);
    void UpdateUserInfo(UserInformation userInfo);
}
```

5. Implement the interface in the view (user control).

```
[SmartPart]
public partial class BrowserView : UserControl, IBrowserView
{
    // ...
   // Generated code
   // ...

    public void UpdateBrowser(string url)
    {
        MainBrowser.Navigate(url);
    }

    public void UpdateUserInfo(UserInformation userInfo)
    {
        userInformationBindingSource.SuspendBinding();
        userInformationBindingSource.DataSource = userInfo;
        userInformationBindingSource.ResumeBinding();
    }
}
```

6. (Optional) Add state properties (typically to the presenter) as outlined in Checklist 14.

```
public class BrowserViewPresenter : Presenter<IBrowserView>
{
    private Dictionary<string, string> _BrowserHistory;

    [State(BrowserWIController.BrowserHistoryStateName)]
    public Dictionary<string, string> BrowserHistory
    {
        get { return _BrowserHistory; }
        set { _BrowserHistory = value; }
    }
```

```
    // ...
    // Other Code
    // ...
}
```

7. Add code for initializing the presenter and the view (via the presenter) in the **OnViewReady** method of the presenter.

```
public override void OnViewReady()
{
    base.OnViewReady();

    // Now create the user information and initialize the view
    WindowsIdentity wid = WindowsIdentity.GetCurrent();

    UserInformation ui = new UserInformation();
    ui.UserName = wid.Name.Substring(wid.Name.IndexOf("\\") + 1);
    ui.Domain = wid.Name.Substring(0, wid.Name.IndexOf("\\"));
    ui.AuthenticationType = wid.AuthenticationType;
    ui.IsSystem = wid.IsSystem;

    // Call view through previously defined interface
    View.UpdateUserInfo(ui);
}
```

8. Add methods with the UI logic to the presenter. Typically, these are for modifying state and/or publish events to notify other components of the state change. You just need to add logic to the presenter if it is really more complex. If it is just about calling one method of another control without any additional logic, such as modifying state or raising events, this is not necessary.

```
internal void OnNavigationComplete(string url, string title)
{
    // Update the state
    if (!BrowserHistory.ContainsKey(url))
    {
        BrowserHistory.Add(url, title);
    }

    // Throw events or update state
    // ...

}
```

9. Add Windows Forms event handlers for controls to the view and call the presenter if necessary.

```
private void GoCommand_Click(object sender, EventArgs e)
{
    MainBrowser.Navigate(UrlText.Text);
}

private void MainBrowser_DocumentCompleted(object sender,
                    WebBrowserDocumentCompletedEventArgs e)
{
    _presenter.OnNavigationComplete(
        MainBrowser.Document.Url,
        MainBrowser.Document.Title);
}
```

As you can see in the preceding example, the first event handler does not call the presenter because the logic is really simple and is just about updating one single other control, whereas

the logic on the **DocumentCompleted** event is more complex and, therefore, the presenter is called in this case.

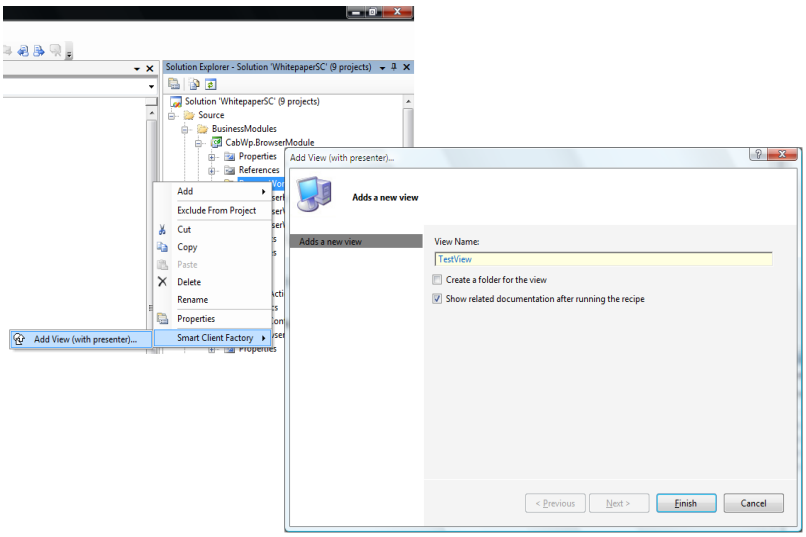


Figure 12: Adding a View with a Presenter

Next, you need to instantiate the SmartParts and activate them when the WorkItem gets activated. You don't need to do anything about the presenter because it gets automatically created by the dependency injection of ObjectBuilder and CAB.

Checklist 16: Instantiating and Activating SmartParts

1. Add instance variables to the **WorkItemController** class working with the SmartParts.

```
public class BrowserWIController : WorkItemController
{
    private BrowserView MyBrowser = null;
    private BrowserHistoryView MyBrowserHistory = null;

    // ...
}
```

2. In the **WorkItemController**'s **RunWorkItem** method introduced in Checklist 12, you need to instantiate the SmartParts using CAB's factory methods. Optionally, when calling the **AddNew** method, you can pass a unique ID for the SmartPart as a parameter if your WorkItem works with more parts of the same type.

```
public void RunWorkItem()
{
    // Get the shell
    MyShellService = WorkItem.Services.Get<IShellExtensionService>();

    // Add state to the WorkItem
    WorkItem.State["BrowserHistoryList"] =
        new Dictionary<string, string>();

    // Now instantiate the SmartParts (and sub-WorkItems)
    if (MyBrowser == null)
    {
```

```
            MyBrowser = WorkItem.SmartParts.AddNew<BrowserView>
                                (
                                    Guid.NewGuid().ToString()
                                );
        }
        if (MyBrowserHistory == null)
        {
            MyBrowserHistory =
                WorkItem.SmartParts.AddNew<BrowserHistoryView>
                                (
                                    Guid.NewGuid().ToString()
                                );
        }
    }
```

3. Next, when the WorkItem will be activated, you need to show the SmartParts in workspaces passed either as parameters, retrieved through the Workspaces collection of the WorkItem, or through the previously introduced **IShellExtensionService**.

```
public void ActivateWorkItem()
{
    // Create a SmartPartInfo
    SmartPartInfo info = new SmartPartInfo();
    info.Title = "Browser opened at " +
        DateTime.Now.ToShortTimeString();

    // Show instances of SmartParts in Workspaces
    MyShellService.ShowInWorkspace
    (
        MyBrowser, info, ShellWorkspaceType.DetailsWorkspace
    );

    MyShellService.ShowInWorkspace
    (
        MyBrowserHistory, ShellWorkspaceType.NavigationWorkspace
    );
}
```

Now that you have implemented all parts of the UI with their presenters and added them to the WorkItem, you need a way to launch a WorkItem. Typically, you would reserve a command for this purpose, which will be invoked through a UIExtensionSite.

Create a Command for Launching First-level WorkItems

CAB provides a ready-to-use infrastructure for working with the Command pattern. You can either create separate classes with a handler implemented or mark methods of your WorkItemController with the [Command] attribute. Checklist 17 outlines the steps to add a new command to a CAB-based component (WorkItem, Controller, or Presenter).

Checklist 17: Defining a Command and Creating a Command-Handler

1. Add a constant to the *CommandNames.cs* file in the *Constants* folder of the interfaces library for your module (*YourModuleName.Interface*).

```
namespace CabWp.TestClient.Modules.BrowserModule.Interface
{
  public class CommandNames :
     CabWp.TestClient.Infrastructure.Interface.Constants.CommandNames
```

```
    {
        public const string LaunchBrowser = "LaunchDefaultBrowser";
        public const string AddNewBrowser = "LaunchNewBrowser";
        public const string CloseBrowser = "CloseCurrentBrowser";
  }
}
```

2. Add a command handler method to the class implementing the command's logic (for example, **WorkItemController** or a Presenter class).

```
[CommandHandler(CommandNames.LaunchBrowser)]
public void CreateDefaultBrowser(object sender, EventArgs e)
{
    ControlledWorkItem<BrowserWIController> NewBrowser;

    // Launch the default browser
    if (WorkItem.WorkItems.Count == 0)
    {
        // Create a new workitem
        NewBrowser =
           WorkItem.WorkItems.AddNew<ControlledWorkItem<BrowserWIController>>(
                                                          "DefaultBrowser");
        NewBrowser.Controller.RunWorkItem();
    }

    NewBrowser = WorkItem.WorkItems["DefaultBrowser"]
                    as ControlledWorkItem<BrowserWIController>;
    NewBrowser.Controller.ActivateWorkItem();
}

[CommandHandler(CommandNames.AddNewBrowser)]
public void CreateNewBrowser(object sender, EventArgs e)
{
    ControlledWorkItem<BrowserWIController> NewBrowser;

    // Generate a new ID
    string WiId = string.Format("BrowserWorkItem#{0}",
                            WorkItem.WorkItems.Count - 1);

    // Create a new workitem
    NewBrowser =
      WorkItem.WorkItems.AddNew<ControlledWorkItem<BrowserWIController>>(WiId);
    NewBrowser.Controller.RunWorkItem();
    NewBrowser.Controller.ActivateWorkItem();
}

[CommandHandler(CommandNames.CloseBrowser)]
public void CloseActiveBrowser(object sender, EventArgs e)
{
    // Close the active browser-workitem
    WorkItem awi = null;
    foreach (KeyValuePair<string, WorkItem> wi in WorkItem.WorkItems)
    {
        if (wi.Value.Status == WorkItemStatus.Active)
        {
            awi = wi.Value;
            break;
        }
    }

    // Terminate the workitem
    if (awi != null)
    {
        awi.Deactivate();
```

```
            if(awi.Status == WorkItemStatus.Inactive)
                awi.Terminate();
        }
    }
```

Declarative CAB command handlers are .NET Framework event handlers. Therefore, the signature of the command handler methods adheres to the standard .NET Framework event handler method signature.

Create an ActionCatalog for Creating UIExtensionSites

The previously created commands are responsible for launching new WorkItems. But they need to be invoked somehow. Therefore, the next step is to create controls on UIExtensionSites. Because you want to create them based on the user's security (some commands are available only to users in particular roles), SCSF supports so-called action catalog classes where you can define actions that are automatically called by the infrastructure if a user is in a specific role.

Checklist 18: Preparing Your Solution for Using the ActionCatalogService When Using Enterprise Library

1. Copy Microsoft.Practices.EnterpriseLibrary.Security.dll to the Lib directory in the root directory of your solution.
2. Add a reference to Microsoft.Practices.EnterpriseLibrary.Security.dll in the *Interface.Library* assembly.
3. Now add a new class to *Infrastructure.Library* that implements the **IActionCondition** interface and is used for validating if a user is permitted to execute an action.

```
public class EnterpriseLibraryAuthorizationActionCondition
                                                : IActionCondition
{
    private IAuthorizationProvider _authzProvider;

    public EnterpriseLibraryAuthorizationActionCondition()
    {
        _authzProvider =
            AuthorizationFactory.GetAuthorizationProvider();
    }

    public EnterpriseLibraryAuthorizationActionCondition(string module)
    {
        _authzProvider =
            AuthorizationFactory.GetAuthorizationProvider(module);
    }

    public bool CanExecute(string action,
                    WorkItem context, object caller, object target)
    {
        try
        {
            return _authzProvider.Authorize(
                        Thread.CurrentPrincipal, action);
        }
        catch (InvalidOperationException)
        {
            // rule (action) not found
            return true;
        }
    }
}
```

4. In the *ShellApplication* class of the *Shell* project of your solution, override the *AddBuilderStrategies* method and add an ObjectBuilder strategy for the ActionCatalog (which has already been generated by SCSF when creating the project).

```
protected override void AddBuilderStrategies(
            Microsoft.Practices.ObjectBuilder.Builder builder)
{
    base.AddBuilderStrategies(builder);
    builder.Strategies.AddNew<ActionStrategy>(
                        BuilderStage.Initialization);
}
```

5. Now override the *AddServices* method in the *ShellApplication* to add the previously created **IActionCondition** rule to your **IAuthorizationService**.

```
protected override void AddServices()
{
    base.AddServices();

    IActionCatalogService actionCatalog =
        RootWorkItem.Services.Get<IActionCatalogService>();
    actionCatalog.RegisterGeneralCondition(
        new EnterpriseLibraryAuthorizationActionCondition());
}
```

6. Next, you can configure authorization rules in your *app.config* file, as explained in the section "Configure the Application."

Checklist 19: Adding an Action Catalog for Registering UIExtensionSites

1. Add a new class to the module's class named *ModuleNameActionCatalog*.
2. Add a new source file with constants for the actions to the *Constants* folder of your module named *ActionNames*.
3. For each action you would like to support, add a constant with the name of the action. This name is used for security configuration purposes of the module.

```
public class ActionNames
{
    public const string BrowseAction = "BrowseAction";
    public const string NewBrowserAction = "NewBrowserAction";
}
```

4. For each action you would like to support, add a method that adds the UIExtensionSite and enables the appropriate command. The signature of an action method needs to have two parameters, one for passing in the caller and another one for passing in the target.

```
public class BrowserActionCatalog
{
    private WorkItem MyWorkItem;
    private IShellExtensionService MyShell;

    public BrowserActionCatalog(WorkItem wi)
    {
        MyWorkItem = wi;
        MyShell = wi.Services.Get<IShellExtensionService>();
    }
```

```
    [Action(ActionNames.BrowseAction)]
    public void BrowseAction(object caller, object target)
    {
        // Allow launching a single browser
        Command Browse =
            MyWorkItem.Commands[CommandNames.LaunchBrowser];
        Browse.Status = CommandStatus.Enabled;

        // Now add the new UIExtensionSite
        MyShell.AddNavigationExtension(
            "Launch Browser",
            Resources.HomeHS,
            Browse);
    }

    [Action(ActionNames.NewBrowserAction)]
    public void NewBrowserAction(object caller, object target)
    {
        // Get all commands and enable them for this action
        // ...

        // Register all UIExtensionSites for this action
        // ...
    }
}
```

5. Disable all commands by default in the *Run*-method of the *ModuleController* and then add the previously added action catalog. After that, register the ActionCatalog created in step 4 and execute necessary actions for initialization.

```
public override void Run()
{
    AddServices();
    ExtendMenu();
    ExtendToolStrip();
    AddViews();

    DisableCommands();
    ExecuteActions();
}

private void DisableCommands()
{
    foreach (KeyValuePair<string, Command> cmd in WorkItem.Commands)
    {
        cmd.Value.Status = CommandStatus.Disabled;
    }
}

private IActionCatalogService _ActionCatalog;
[ServiceDependency]
public IActionCatalogService ActionCatalog
{
    get { return _ActionCatalog; }
    set { _ActionCatalog = value; }
}

private void ExecuteActions()
{
    // Create the action-catalog
    WorkItem.Items.AddNew<BrowserActionCatalog>();
    ActionCatalog.Execute(ActionNames.BrowseAction,
```

```
                                this.WorkItem, this, null);
        ActionCatalog.Execute(ActionNames.NewBrowserAction,
                                this.WorkItem, this, null);
    }
```

The preceding example immediately called all actions when the **ModuleController** of the module was initialized. But that is not necessary. For example, think about actions being executed as part of command handlers. For example, the commands introduced in the section "Create a Command for Launching First-level WorkItems" directly implement the logic. But you can also create actions for the logic executed in commands instead of directly executing the logic in the command handlers. Actually, from a security perspective, this is the preferred way for larger solutions.

Publish and Receive Events

Next, components loaded into a CAB-based smart client can use event publications and event subscriptions to exchange data between them. The event broker of CAB is intended for building loosely coupled events—that means the publisher does not know about any subscribers and vice-versa. The relationship between publisher and subscribers is ensured through event URIs, which are uniquely identifying events. The following checklists outline the steps for creating an event publication and event subscription.

Checklist 20: Create an Event-Publication Using SCSF

1. Add a class for the event arguments to the interfaces project of your module.

```
public class BrowserEventArguments : EventArgs
{
    private string _Url;
    public string Url
    {
        get { return _Url; }
    }

    private string _Title;
    public string Title
    {
        get { return _Title; }
    }

    public BrowserEventArguments(string url, string title)
    {
        _Url = url;
        _Title = title;
    }
}
```

2. Switch to the actual project that implements the parts of the module. Right-click the class that should publish the event and use the *Smart Client Software Factory* context menu to add an event publication.
3. Complete the fields in the wizard, as shown in Figure 13, by specifying your event URI constant and your event arguments class. Also decide about the publication scope; keep it as narrow as possible. If an event is just required within the current WorkItem and its sub-WorkItems, select a PublicationScope of WorkItem. Global is the default for this option and means the event will

be published to all parts of the smart client application. This automatically generates code similar the following code.

```
[EventPublication(
    EventTopicNames.CabWpBrowserNavigated, PublicationScope.Global)]
public event EventHandler< BrowserEventArguments> CabWpBrowserNavigated;

protected virtual void OnCabWpBrowserNavigated(BrowserEventArguments eventArgs)
{
    if (CabWpBrowserNavigated != null)
    {
        CabWpBrowserNavigated(this, eventArgs);
    }
}
```

4. Now you can raise the event when necessary without knowing who has subscribed to the event.

```
internal void OnNavigationComplete(string url, string title)
{
    // Throw events or update state
    OnCabWpBrowserNavigated(
        new BrowserEventArguments(url, title)
    );

    // Update the state
    if (!BrowserHistory.ContainsKey(url))
    {
        BrowserHistory.Add(url, title);
    }
}
```

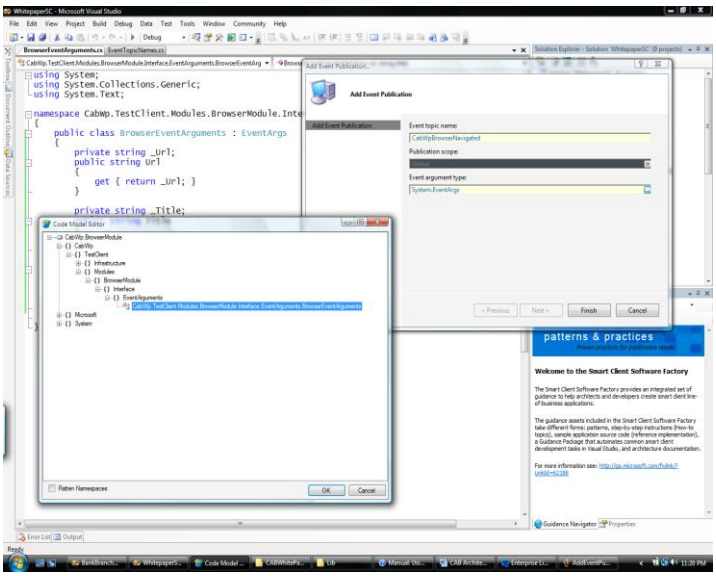


Figure 13: Adding an Event Publication Using SCSF

Next, you can subscribe to the event using the *Smart Client Software Factory* context menu on the component (such as WorkItem, Presenter, or Controller) you want to subscribe to the event. Checklist 21 describes how to do this.

Checklist 21: Create an Event Subscription Using SCSF

1. Switch to the class that needs to subscribe to an event.
2. Right-click this class, and then click *Smart Client Factory – Add Event Subscription*.
3. In the wizard, select the event subscription from the drop-down menu, and then select the event arguments required for a specific event subscription, as shown in Figure 14.
4. The wizard generates an empty event handler method, which will be automatically connected with the publisher. In this method, you can now add the required logic for it to be executed when the subscribed event is received.

```
[EventSubscription(...)]
public void OnCabWpBrowserNavigated(object sender,
                                    BrowserEventArguments eventArgs)
{
    View.AddHistoryEntry(eventArgs.Url,  eventArgs.Title);
}
```

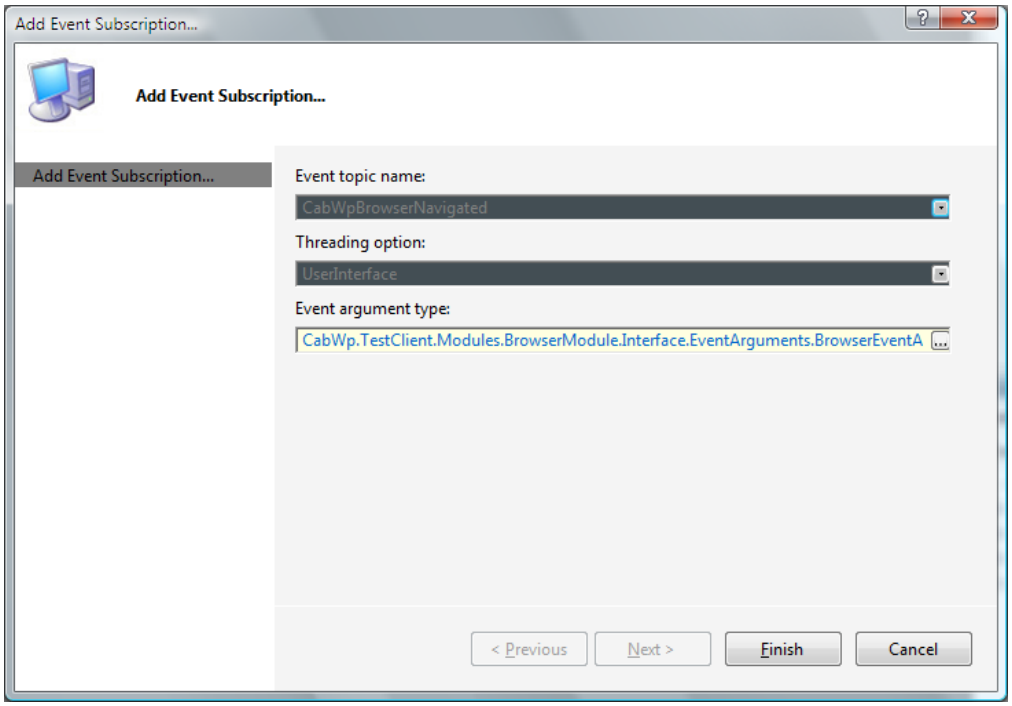


Figure 14: Adding Event Subscriptions Using SCSF

Configure the Application

Finally, you need to configure your application. Primarily, there are two parts you need to configure: the catalog containing the list of modules to be loaded and the application configuration itself. When using SCSF, the profile catalog understands dependencies between modules. You need to configure all modules you want to load for your application as the following example demonstrates.

```
<SolutionProfile xmlns="http://schemas.microsoft.com/pag/cab-profile/2.0">
  <Section Name="Layout">
    <Modules>
      <ModuleInfo AssemblyFile="Infrastructure.Layout.dll" />
    </Modules>
  </Section>
```

```
  <Section Name="Services">
    <Dependencies>
      <Dependency Name="Layout" />
    </Dependencies>
    <Modules>
      <ModuleInfo AssemblyFile="Infrastructure.Module.dll" />
      <ModuleInfo AssemblyFile="CabWp.Messaging.dll" />
    </Modules>
  </Section>
  <Section Name="Apps">
    <Dependencies>
      <Dependency Name="Layout" />
      <Dependency Name="Services" />
    </Dependencies>
    <Modules>
      <ModuleInfo AssemblyFile="CabWp.BrowserModule.dll" />
    </Modules>
  </Section>
</SolutionProfile>
```

For more information about the solution profile, see the documentation for CAB and SCSF. Next, you need to modify the configuration of your app.config file– especially for the action catalog covered in the section "Create an ActionCatalog for Creating UIExtensionSites." Checklist 22 lists the steps.

Checklist 22: Configure the Action Catalog for Enterprise Library

1. Open app.config of the *Shell* project.
2. Register a new configuration at the very top of the app.config file, as follows.

```
<configSections>
  <section name="loggingConfiguration" type="..." />
  <section name="exceptionHandling" type="..." />
  <section name="securityConfiguration" type="...Security..." />
</configSections>
```

3. Configure your authorization provider and the rules as needed.

```
<securityConfiguration
    defaultAuthorizationInstance="WhitepaperSC_Security"
    defaultSecurityCacheInstance="">
      <authorizationProviders>
        <add type="...AuthorizationRuleProvider..."
             name="WhitepaperSC_Security">
            <rules>
              <add expression="R:Everyone" name="BrowseAction" />
              <add expression="R:Everyone" name="NewBrowserAction" />
            </rules>
        </add>
      </authorizationProviders>
</securityConfiguration>
```

Finally, you can now run and test the application.

Summary

This white paper provides guidelines for designing and implementing smart clients based on Composite UI Application Block and Smart Client Software Factory. In the first part, you learned about architectural guidance for helping you with decisions for identifying CAB-based components, such as WorkItems and services, and decide how to package them into modules. You learned about the categorization of services in UI-related shell services and infrastructure services. You learned about a use case–driven strategy and an entity-driven strategy for identifying WorkItems, and you learned about criteria for packaging WorkItems into modules together.

The second part is guidance for your developers based on SCSF. It consists of a number of checklists you can give your developers to help them quickly ramp-up with CAB and SCSF after they work through the basic documentation.

Figure Index

Checklist Index