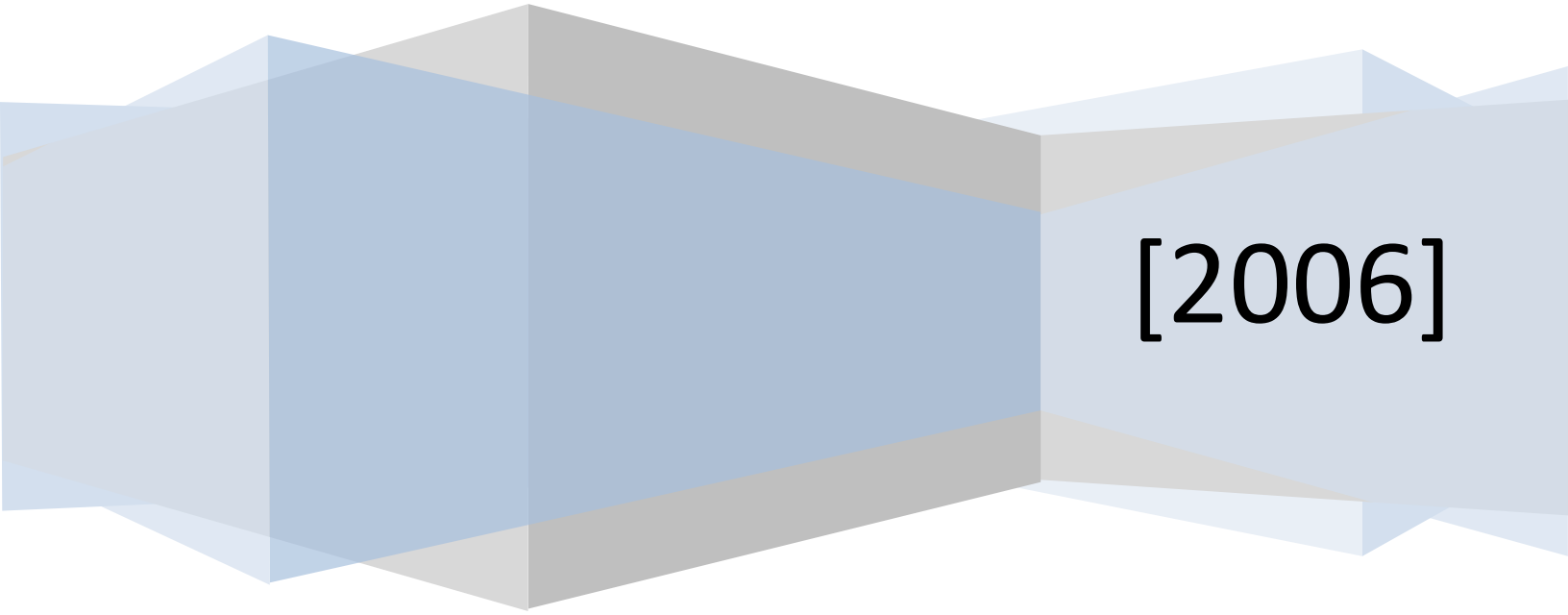
Microsoft

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 whitepaper, originally created for the Austrian bank Raiffeisen Linz, RACON/GRZ, provides architectural guidance for designing and implementing composite smart clients based on the Composite UI Application block and the Software Factory for Smart Clients provided by the Microsoft patterns and practices group.

The original intention was supporting the architects and lead developers of Raiffeisen Linz 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 Raiffeisen has lots of different client-based applications to support bank-employees in their daily business. Primarily the problem was, that these each of these applications worked differently and the employees really have to struggle with the fact, that the user experience is different across these applications. Furthermore the applications are not integrated at all resulting in many tasks being accomplished multiple times in different applications. From the IT-perspective, the management of that many applications is fairly hard and the reuse of common functionality is nearly impossible. Therefore Raiffeisen 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 bank-application, should be loaded and initialized dynamically based on the user's security-context so that users see the parts they are allowed to work with, only. Of course a common infrastructure such as a bank-desktop can also manage common services used by all the different applications plugged in dynamically, in a central infrastructure. This makes management easier and allows re-usage of common client-side services. As the modules are dynamically configurable, one of the core requirements was to enable the components to communicate in a loosely coupled fashion so that the modules do not really depend on each other. Finally the Bank-Desktop composite application was born. It provides a common client-application infrastructure for every bank-employee of Raiffeisen Linz. Each client-based application of Raiffeisen Linz will be integrated into this bank-desktop by a role-based configuration suited to the tasks of an employee of a specific role. The bank-desktop therefore is a very good example of a very special flavor of smart clients – so called *composite smart clients*. Microsoft's offering for designing and developing composite smart clients is made up by Composite UI Application Block and Smart Client Software Factory. This article is all about these technologies.

Scope of this Whitepaper

The primary target of this whitepaper is providing architectural guidance on developing composite smart clients based on Composite UI Application Block (aka CAB) and the Smart Client Software Factory (aka SCSF). Therefore this whitepaper helps architects with the following decisions:

- Structuring project teams as CAB & SCSF affects the development process of smart clients
- Identification of the primary components for composite smart clients
- Packaging of these components into deployment-units
- Usage of infrastructure components such as the event-broker and commands
- Shell design decisions

The second part of the whitepaper contains cook-book like instructions for using CAB and SCSF. These instructions can be provided to developers for a faster ramp-up when start working with both, CAB and SCSF. This whitepaper is **neither a detailed documentation on the APIs of Composite UI Application Block nor a detailed description on the guidance automation packages provided by Smart Client Software Factory**. For further details on CAB and SCSF refer to the following MSDN articles:

- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/cab.asp
- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/scsflp.asp
- http://msdn.microsoft.com/msdnmag/issues/06/09/SmartClients/

Requirements for reading this Whitepaper

For completely understanding this whitepaper you need to know about the basic structure of Composite UI application block. Understanding Smart Client Software Factory is optional for the first part of the whitepaper and is required for the second part of the whitepaper. For detailed information on CAB and SCSF please refer to the links provided in the section “Scope of this Whitepaper”.

To test and follow the samples introduced in this whitepaper, you need to install the following components on your development-machine

- .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 higher
  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, May 2006
  http://www.microsoft.com/downloads/details.aspx?FamilyId=7B9BA1A7-DD6D-4144-8AC6-DF88223AEE19&displaylang=en
- Guidance Automation Extensions, June 2006
  http://www.microsoft.com/downloads/details.aspx?familyid=C0A394C0-5EEB-47C4-9F7B-71E51866A7ED&displaylang=en
- Guidance Automation Toolkit, June 2006
  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

In this first part the whitepaper covers architectural guidance for creating composite Smart Clients based on 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 will drill down into the guidelines provided by this first part, you will need to make sure to understand basic terms used throughout the whitepaper. This section summarizes these terms.

Composite UI Application Block (CAB)

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) 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- 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 which 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* knowing all the different aspects of a use case. As such, a WorkItem of course 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 which are common for the whole client application, for a specific module or just for WorkItems within a certain hierarchy of WorkItems (e.g. 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 such 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 specifying which modules and services need to be loaded into the application. By default the catalog is just an XML-file residing 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 (see next table-entry).
ModuleInit	Each module consists of a ModuleInit class which is responsible for loading all the services and WorkItems (as well as other aspects such as user interface extensions to the shell – see UIExtensionSite) of a Module.
Shell Application	The Shell-Application is the primary application which 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 as well as starting the main form (Shell).
Shell	This is the main-form of the application providing the user interface which 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 as well as WorkItems created and registered by these modules.
Workspace	A workspace is a control which 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 which are going to be dynamically filled with UI provided by WorkItems or sub-WorkItems. Any user control can be a workspace by implementing the IWorkspace interface. CAB out-of-the-box comes with classic container controls as Workspaces such as a tabbed work space.
UIExtensionSite	UIExtensionSites are special placeholders for extending fixed parts of the shell such as menus, tool-strips or status-strips. Opposed to Workspaces they are intended to be used for parts of the UI, which should not be overridden completely by WorkItems but should just be extended (such as adding a new menu entry).
Command	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 method which 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 events which 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 as publication marked with the [EventSubscription] attribute.
EventSubscription	Opposite to EventPublication. Any class that wants to subscribe to an event with a specific event-URI needs to implement an event-handler matching 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 throws an event with a matching event-URI and method-signature.
Model	The (client-side) business entity a WorkItem will process, e.g. Customer, BankAccount or Loan.
View	A View is an ordinary .NET User Control which 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 UI-logic only, whereas the related client-business logic is implemented in the presenter/controller.
SmartPart	See view! A smart part is a .NET 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	A class implementing the logic for modifying a model. One controller can modify a model presented by many views. Origin: Model-View-Controller
ObjectBuilder	A foundation component of CAB acting 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	A pattern which 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 role in CAB is crucial for any good composite smart client design based on CAB. Nevertheless this whitepaper does not focus on the base-classes and APIs of CAB, therefore for more information take a look at the CAB documentation or at the MSDN articles referenced in the section "Scope of this Whitepaper" earlier.

Smart Client Software Factory

Although CAB provides a great infrastructure, getting started with CAB is definitely a challenge. Furthermore working with CAB requires developers completing many manual steps such as creating classes inherited from the WorkItem base class to create a use case controller or creating Controller-classes, View-classes and Model-classes manually. The Smart Client Software Factory is an extension to Visual Studio 2005 Professional (or higher) providing added functions for automating these tasks and provides a huge, detailed documentation on creating composite smart clients using CAB including how-to's 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 provided by the Microsoft Patterns & Practices team as well. The Guidance Automation Extensions are an infrastructure allowing Architects and Developers easy creation of extensions into Visual Studio for automating typically development tasks with the primary target 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, use case controllers as well as event publications and subscriptions. Of course these guidance packages are completely extensible. This whitepaper assumes that you are familiar with the basic concepts of SCSF and Guidance Automation Extensions. But as there are a couple of terms which 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 WorkItem-Controller is a class introduced by 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 Smart Client Software Factory and 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 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 Smart Client Software Factory automatically creates a WorkItem for a module named ModuleController. The ModuleController therefore is the primary entry-point for all WorkItems provided by a module.
Presenter	A class implementing 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 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 as well as the model can update the view.

For more information on SCSF and Guidance Automation Extensions refer to the following MSDN articles:

http://msdn.microsoft.com/vstudio/teamsystem/workshop/gat/default.aspx

http://msdn.microsoft.com/vstudio/teamsystem/workshop/gat/intro.aspx

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 supporting developers to create CAB-based smart clients with minor effects on architectural decisions – it just 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 Composite UI application block is very use case centric. Therefore start having a good understanding of your use cases (in context of the client-application) is crucial. Primarily you can think of three aspects affecting 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 into the shell directly or at least affecting the shell's layout and design. Non-UI related functionality which 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 either using the command-pattern or the event-broker sub-system of CAB. Logically close 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 refactor your WorkItem packaging slightly. 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. First-level WorkItems are such WorkItems, which don't have a parent WorkItem other than the RootWorkItem of the shell (or the ModuleController-WorkItem introduced by Smart Client Software Factory).

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

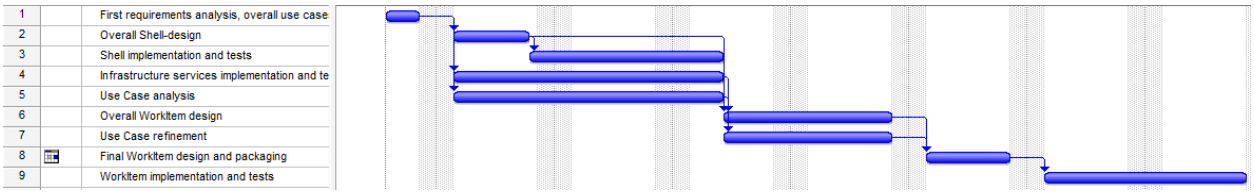


Figure 1: Development process outlined – schematically

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 as well as infrastructure services. First and foremost it is essential that the team identifies the core requirements to these infrastructure services and the shell. These requirements needs 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 can start with a detailed use case analysis together with the business departments. In the meantime members of the core team can start developing the shell as well as the necessary infrastructure services. The shell as well as infrastructure services are implemented one time

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

As 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 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 figure out some minor new requirements on the shell and the infrastructure services that require you changing the service interfaces (of course building a prototype before starting the actual project probably makes sense in your case as well – but that's finally 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 via commands. Typically a refinement and a detailed analysis of the relationships between use cases are indicators of which type a relationship is (parent-child, event, command). This refinement will affect your strategy in packaging WorkItems as well. Therefore it is essential to keep this refinement-iteration in mind before start developing the broad range of WorkItems. You will get more details on each of these steps in the following sections of this whitepaper.

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 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 building your own business-application framework on top of CAB, which is composed by your shell and infrastructure services as shown in Figure 2.

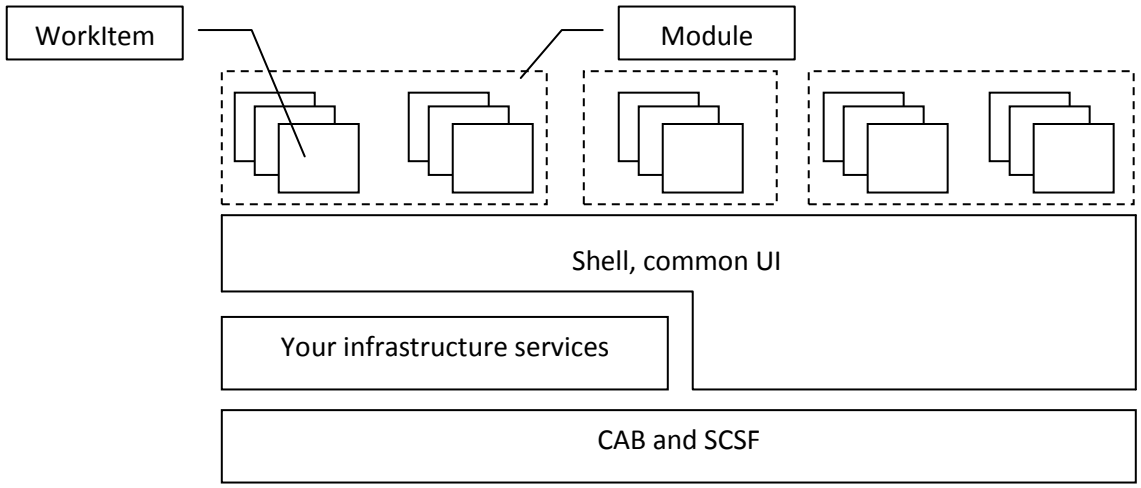


Figure 2: Layering of CAB-based Smart Clients

Application Shell- and Infrastructure-Service Design

One of the first you will design and implement for your CAB-based Smart Client is the shell. The shell is the actual application which is responsible for loading and initializing base client-services, loading and initializing modules as well as 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 need to be added to the shell are:

- Menus, toolbars and tool-strips
- Navigation-controls such as the Outlook-bar 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, warnings etc.)
- Task panes such as the ones you know from Office 2003/2007 or task bar you know from the Windows XP Explorer
- Context-panes which are displaying 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 an 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.

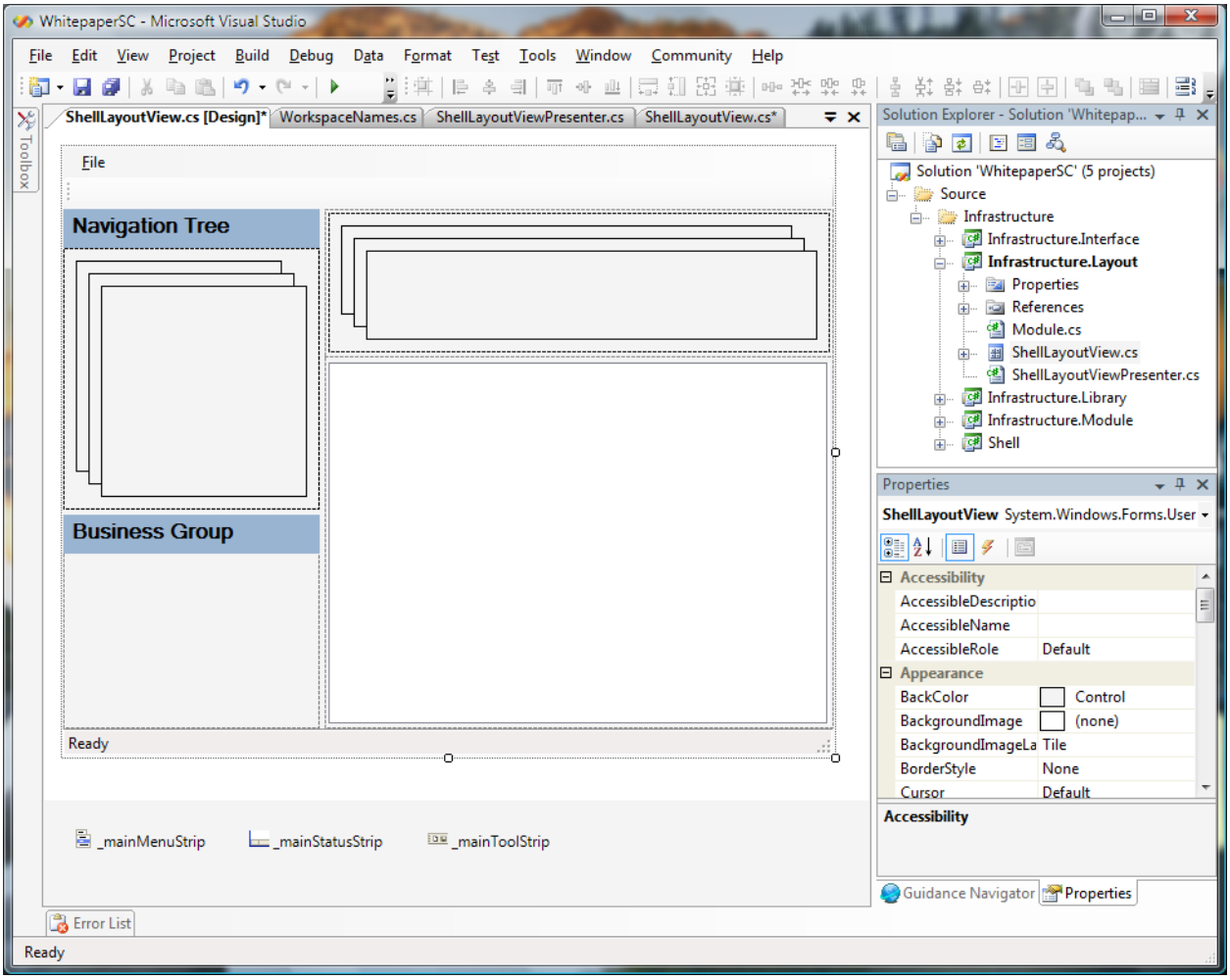


Figure 3: A typical example for a shell with Workspaces and UIExtensionSites

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 a maximum of flexibility you might want your developers having a "strongly-typed" way of accessing the shell. Furthermore this introduces a very tight coupling to the contents of the shell. Therefore I recommend a layer of indirection between the shell and components that want to add 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:

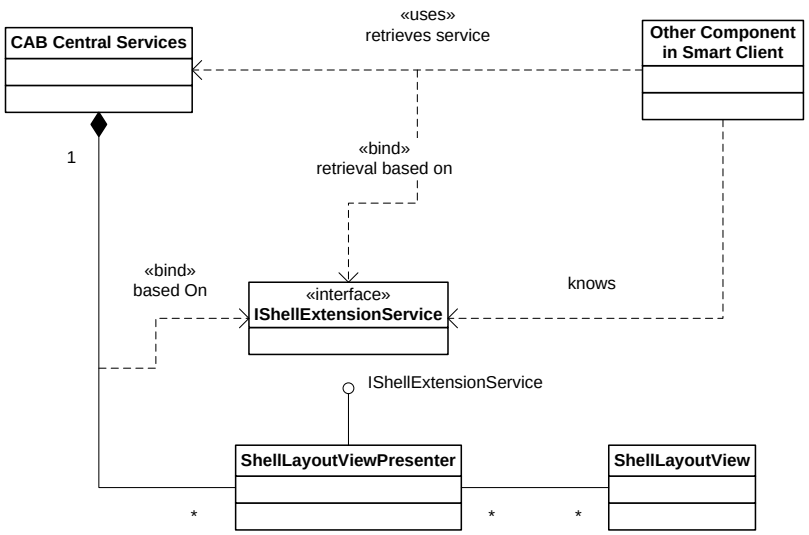


Figure 4: Shell implementing you custom IShellExtension interface provided as a central service

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- 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);
```

Still the implementation of this IShellExtension interface can 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 task bar 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 versus UIExtensionSite

As mentioned earlier, workspaces are used for replacing a complete part of the shell with a different UI. This is typically 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) and therefore in most cases they are added by ModuleInit components.

Direct integration in the Shell versus 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 in other Smart Client shells? Or maybe in other modules 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 you are thinking about being integrated directly into the shell or if you need to load it based on the end-user's security context, then you should put this common UI into a separate module as well.

Putting the shell's layout in a separate module or directly in the application

Of course you can put the shell's main-view with the overall layout into a separate module as well. Actually the Smart Client Software Factory asks you this question when creating a new project. But how-to decide on this? Well, again it's fairly easy: if you want to exchange the shell's layout dynamically based on the security-configuration or the common configuration of your Smart Client, then you need 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

Definitely you will ask now, what happens if you decided adding UI directly to the shell or adding the layout directly to the main-application instead of putting it into a separate module but now

requirements are 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 to many) modules and components loaded into the Smart Client. Opposed to shell-infrastructure, these services are not bound to UI-specific tasks. Much more they encapsulate client-side logic (maybe client-side business logic for offline capability). CAB out-of-the-box 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 introduce your own services as well. Typical examples for central infrastructure services not introduced by CAB are:

- Authorization service (CAB and SCSF do not provide one, out-of-the-box, but the BankBrenchWorkbench 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 (see section ”Context and WorkItems” later in this whitepaper).
- 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 as CAB allows you registering 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 loaded before other modules with actual use case implementations will be 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 as 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 manages state required across SmartParts and sub-WorkItems. Each CAB-Application has one RootWorkItem which acts as central entry point for global services as well as 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 called 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 designing 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 all the required parts for doing so, together. Take a look at the use case diagram shown in *Figure 5: A sample Use Case-diagram*. 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.

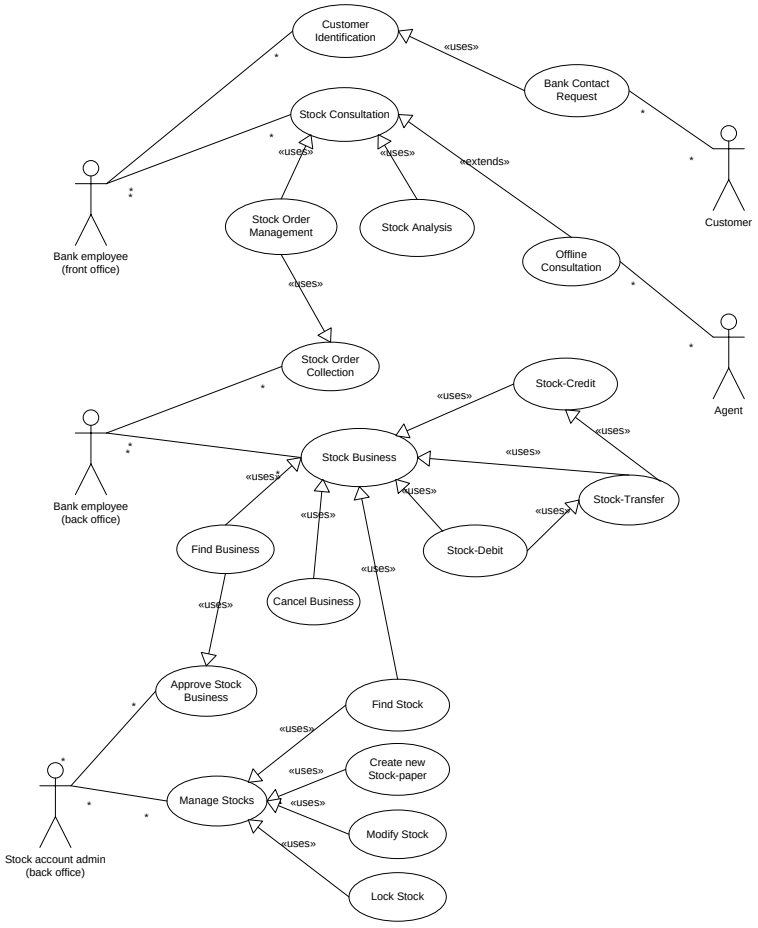


Figure 5: A sample Use Case-diagram

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 flavors: either a use case is 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 which are not accessible from outside their parents whereas 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 implemented in the ModuleController, directly) – 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. I recommend 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 launching into WorkItems 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.

Simple WorkItems without sub-WorkItems or sub-WorkItems themselves typically 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 launching into the first-level-WorkItem if it should not be created automatically.

Exclusion of Use Cases

When taking a closer look at the use case diagram you will figure out, that not all use cases will result in WorkItems. Take a look at the "Bank Contact Request" use case. Remember that for the while you want to design a smart client used by bank employees, only. A customer requesting a stock consultation (expressed through the "Bank Contact Request") can either go to the front desk directly or use an Internet-banking or net-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. Just the opposite, "Customer Identification", needs to be part of the Smart Client you are designing now.

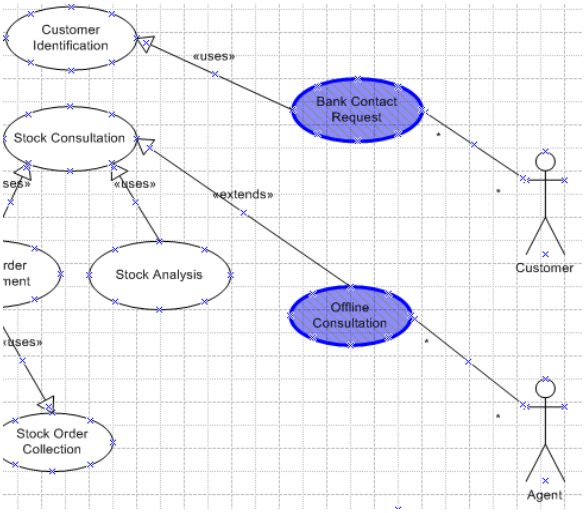


Figure 6: Exclusing Use Cases

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. This is something you therefore need to verify against the infrastructure services identified earlier: web service agents for example 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. This typically happens when a use case logically belongs to a parent-use case but in a detailed analysis you figure out that it does not share state, context or anything else with other sub-use cases and it does not depend on shared state, context or other commonalities as well. In that case for avoiding unnecessary overheads you should make them to first-level-WorkItems as well. 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 just used for finding stock-papers or ordered business-transactions. They do not depend on shared state or on context of the first-level

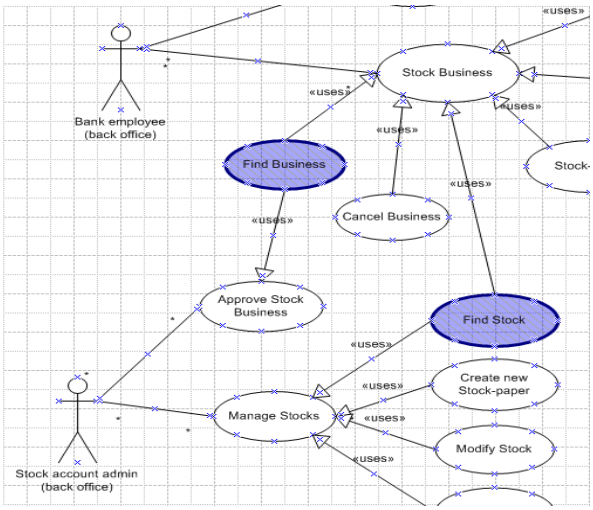


Figure 7: Root-WorkItem or Sub-WorkItem

WorkItem. So they are perfect candidates for becoming first-level-WorkItems themselves called through commands registered by the module which they are part of itself.

WorkItem-Identification – Summary

To summarize, the steps for identifying WorkItems based on use case diagrams for CAB-based composite Smart Client are the following ones:

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 candidates for first-level-WorkItems as well.

The 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 resulting from the use cases of figure 5.

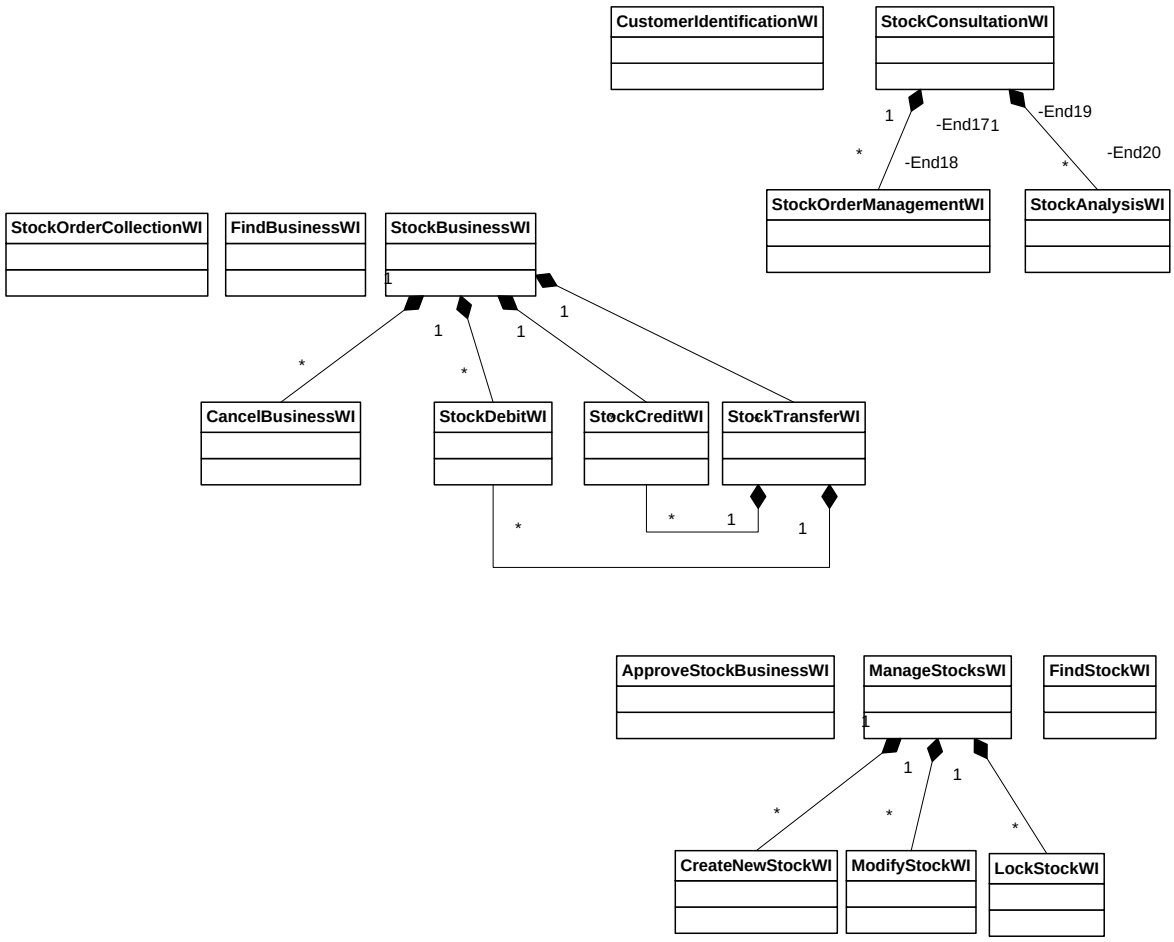


Figure 8: Class Diagram showing the WorkItems for the previous use cases

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 are Customer, Product, Stock, StockCredit, StockDebit or StockTransfer. For each of these entities you create a WorkItem. As StockTransfer is a combination of both it uses StockCredit and StockDebit as sub-WorkItems.

Packaging WorkItems into Modules

After you have identified 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 addressing the same space of business into a module. Taking the use case diagram from figure 8 that would mean, you create a module for stock consultation, one for stock business WorkItems and a last one for stock management as shown in figure 9.

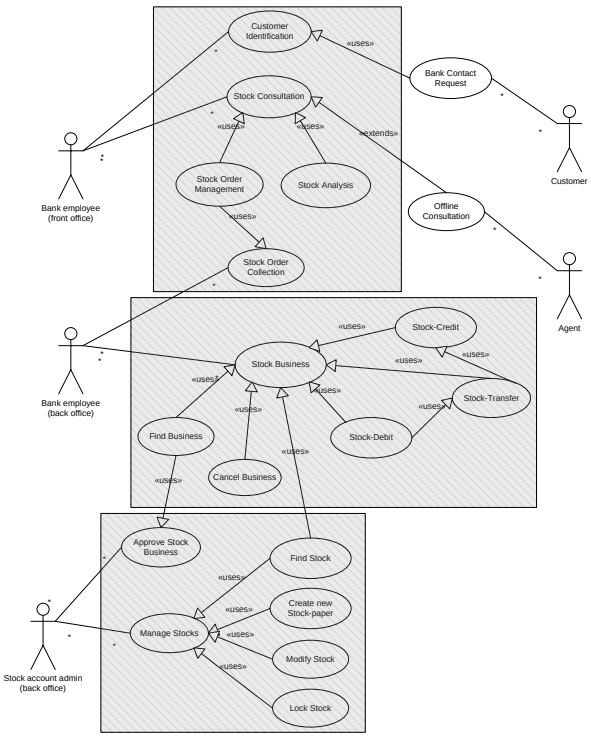


Figure 9: Packaging WorkItems into Modules

But there are some additional criteria for deciding on the packaging of WorkItems into modules. These additional criteria are:

- Security
- Configurability
- Reusability

First and foremost modules are configured in a profile-catalog. This catalog by default 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 on how-to package WorkItems into modules. Suppose a user is not allowed to manage any stocks as shown in the use cases above. But definitely users not allowed creating new stocks or locking 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 in figure 9) in a way that bank employees from the back office are not allowed using it, they are not allowed to use the "Find Stock" WorkItem as well. But as they need this WorkItem as well it is useful packaging 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 encapsulating them into separate modules. If you need to configure WorkItems independently from others you put them into separate modules as well.

In the example demonstrated in figure 9 this is true for “Find Stock”, “Find Business” and “Stock Order Collection” WorkItems. Therefore it is useful encapsulating them into separate modules as shown in figure 10:

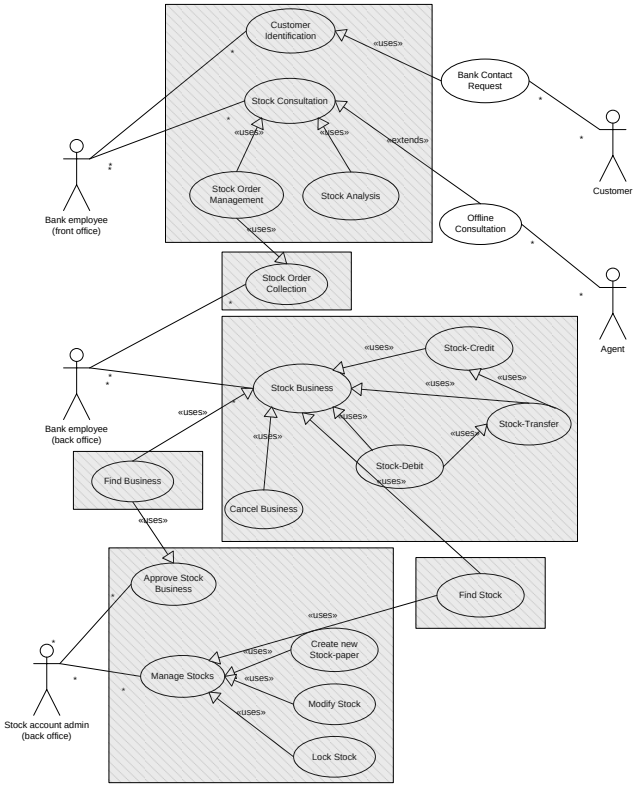


Figure 10: New packaging according to security and reusability requirements

If you figure out that configuration-, security- and reusability-requirements are equal (or nearly the same) for some of the WorkItems outsourced into separate modules according to figure 10, you can package them into one module instead of three modules, as well. For example if the security-, configuration- and reusability-requirements of all “Find X” WorkItems are equal, you can create one module containing all the “Find X” WorkItems.

CAB-Infrastructure Usage-Guidelines

Relationships between use cases are representing interactions between use cases. CAB supports two types of interactions between components in general – active communication through an implementation of the command pattern and passive communication through its event-broker system. But when should you use what?

Using the Event-Broker System

The event-broker of CAB provides you with an 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 identifying an event uniquely.

Therefore, use the event-broker really 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 who receives events, then the event-broker is the right way to go. If you need immediate response or need to know who exactly 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 you component needs something to be done immediately by another component or passing control on to this other component, then the command pattern infrastructure is the right thing. Use commands for launching WorkItems (or sub-WorkItems) instead of events. Use commands to cause them something to do.

Part II: Developer's Cook-Book Guidance

While the first part 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 form of a step-by-step guide. Therefore it can be seen as a cook-book like extension to the guidance provided by CAB and Smart Client Software Factory.

Developer's View on the Development Process

First of all we will outline the necessary development steps for creating a CAB-based Smart Client. This part of the whitepaper assumes that you have all of the requirements mentioned in the section "Requirements for reading this Whitepaper" of "Part I: Architectural Guidance for Composite Smart Clients" installed on your machine. It will use Smart Client Software Factory for completing 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 steps 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 whitepaper. Each of the steps is supported by Smart Client Software Factory to a certain extend. When using plain CAB you need to perform these steps manually.

Create a new CAB/SCSF Project

First you need to create a new Smart Client project using Smart Client Software Factory. For this purpose complete the steps shown 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:
    a. Location of compiled CAB assemblies. Note that you need to compile these manually after you have installed CAB.
    b. Location of Enterprise Library assemblies. They will be compiled automatically when installing Enterprise Library.
    c. The root-namespace for your Smart Client project.
    d. If you want to add the layout of your shell into a separate module. This needs to be done when you want to configure the shell's layout through the configuration as well and you want to be able to change the shell's layout via configuration.

3. Next you click finish and SCSF creates a number of projects for you.

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 SCSF. Table 3 contains detailed descriptions on the projects created by the Smart Client Software Factory for you when completing the "Smart Client Application" wizard.

Table 3: Projects created by 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. This module by default consists of an empty module-controller (which is a default-root-WorkItem for a module). You should only add services here used by the whole Smart Client application that typically loaded always with your application independently of other modules.
Infrastructure.Layout	If you selected that you want to put the shell's layout into a separate module, SCSF creates this project containing a SmartPart and a presenter for this SmartPart for the shell-layout.
Shell	The actual Windows-application of the Smart Client containing the configuration and the actual main-form that hosts the shell-layout.

As mentioned, 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 client's object model to the (web) service's object models for loose coupling between those two.
Basic SmartPart Info Classes	CAB allows you creating classes implementing ISmartPartInfo. You need to implement this interface in a class when you want to display additional information for a SmartPart when 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 that it displays the information. SCSF creates some basic implementations of this interface used frequently.
ActionCatalog Service	Allows you defining specific methods of your classes as actions which 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 a CAB Business Modules".

DependentModuleLoader	By default CAB comes with a simple profile-catalog and module loader for loading modules dynamically based on the configuration of the ProfileCatalog.xml file stored in the application's directory. SCSF adds a new module loader that allows you specifying 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. The DependentModuleLoader is used by default in SCSF-based projects.
WebServiceCatalog Service	This added service allows you retrieving the modules to be loaded at start-up of your Smart Client from a web service. All you need to do is implementing the web service backend 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 start-up project. This is more convenient as by default Visual Studio automatically takes the currently selected project as your start-up project.

Creating the Shell and its Components

The first step after creating the project is designing your shell and the basic components belonging to your shell according to the requirements you have specified earlier in the project life-cycle. The following check-list outlines the processes for creating the shell and it's 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 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 Checklist 3 are covered in detail in the following sections of this section in this whitepaper.

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. Workspaces of CAB can be added via the designer as well.

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 as they will be processed when you come to 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 belonging to the Workspcae you have just deleted in its constructor.

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

5. (optionally) Add the CAB controls to your Visual Studio 2005 controls toolbox.
   a. Open the toolbox, right-click it and 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 as shown in Figure 11.
6. Now you can drag & drop CAB-controls onto your surface as you need them.

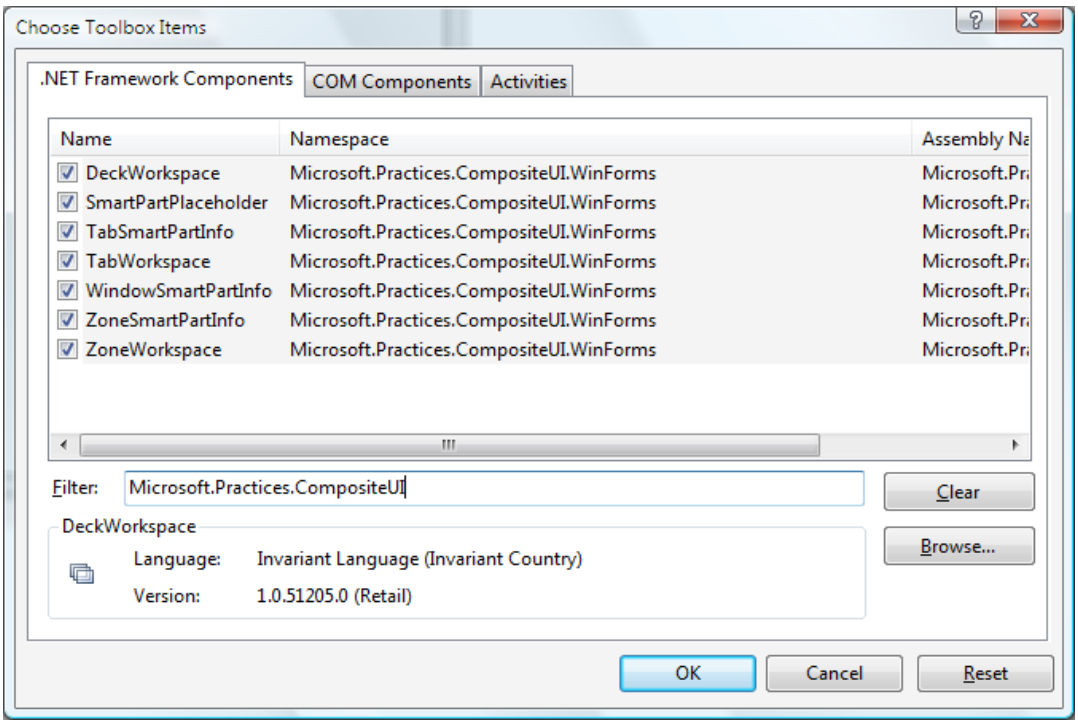


Figure 11: Choose Toolbox Items

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 & 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 containing 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 UIExtesionSites and register them.

Register UIExtensionSites

Opposed to Workspaces UIExtensionSites are fixed parts of the shell which cannot be replaced completely but can be extended by modules loaded into the application. Typical examples are menus, tool-strips or status-strips of the application which are basically defined by the shell but can be extended dynamically as modules are loaded into the application. The basic steps for registering a UIExtensionSite are the following:

Checklist 6: Adding a new UIExtensionSite

1. Add a menu, tool-strip, tool-strip item, status-strip or status-strip item you wish 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 as 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 or not) 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 take into account when registering the site.
    a. 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.
    b. If you want items to be inserted immediately after a specific MenuItem or ToolStripItem you register this item as a UIExtesionSite.
    c. If you want items to be entered just into a ToolStrip, StatusStrip or MenuStrip, you just register this ToolStrip, StatusStrip or MenuStrip as a UIExtensionSite.
6. Registering a UIExtensionSite needs to be done in code. There are two different situations you need to take into account when registering the site.

a. When created 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);
    }
```

b. When created the shell directly in the application, then open the *ShellApplication.cs* source-file in the *Shell* project and add register 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 have registered 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 putting an indirection between UI-parts of the shell and actual WorkItems customizing the shell's UI. Although you can access any UIExtensionSite and Workspace in a generic fashion as follows,

```
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 by referencing non-existent UIExtensionSites and Workspaces. Furthermore it allows you adding 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 creating just one interface and defining 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 that they are available for all subsequently loaded modules. There are three different choices you need to take into account:

- The shell-based service is closely related and tightly coupled to the shell's layout. In that case in 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 as it is not tightly bound to the layout but it will not be reused in other applications. Then it's not necessary putting it into a separate module.
- The shell-based service will be reused in other applications as well. Then it is useful encapsulating it into a separate module. But then furthermore you need to establish a contract between the shell and the shell-based service as well.

The checklist Checklist 9 outlines the steps for creating a shell service tightly coupled to the shell UI. Creation of shell-services explained in the second and third point above are created such as normal foundation-services modules as outlined in the section "Creating and Registering new Services".

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 follows:

```
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 containing general services

1. Add a new *Foundation Module* to the solution (from the Smart Client Factory context menu in your solution).
2. Activate the following options 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 have been created, can access the service as follows:

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

Finally that's it 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, already, and you have exposed the interfaces for these proxies explicitly, all you need to do is creating 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 registering 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 a 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 a such modules are outlined in Checklist 11:

Checklist 11: Steps for creating a CAB Business Module

1. Adding the new Business Module using 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 in the whitepaper.

Adding the new Business Module

First you need to create a new business module via the *Add New Business Module* of the Smart Client 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:

- 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 created when adding a new business module and their role. These are part of the actual business module, the interfaces module just consists of the constants-folder, but not 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 create WorkItems directly, it creates WorkItemControllers implementing 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 some).
Constants-folder	Containing files for defining constants for UIExtensionSites, Workspaces, command-names and event-topics belonging 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 create the WorkItem manually as outlined in Checklist 12.

Checklist 12: Creating a new WorkItem

1. Create a new folder for anything belonging to your WorkItem in the module's project.
2. Add a new class and add the following CAB-related namespaces as well as the namespaces pointing 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 the utility-classes. If you want to throw events, you need the event-broker namespace in addition to the others.

```
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 running, already (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 having now parameters defined – now Workspaces or UIExtensionSites passed in. That means this is a first-level-WorkItem that requires to acquire a reference to the IShellExtensionService introduced in the section “Creating the Shell and its Components” earlier 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 as 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 it is the case for first-level-WorkItems. Sub-WorkItems typically called through first-level-WorkItems only.

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. Optionally create a class for the data stored in your state (not necessary if you have your business-entity = model defined, already or if the class is provided by other parts such as the base class library of the .NET Framework, already).
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 / 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

1. 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 the item is created via ObjectBuilder.

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

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

2. 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 recipe 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 throwing 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 example above, the first event-handler does not call the presenter as the logic is really simple and 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.

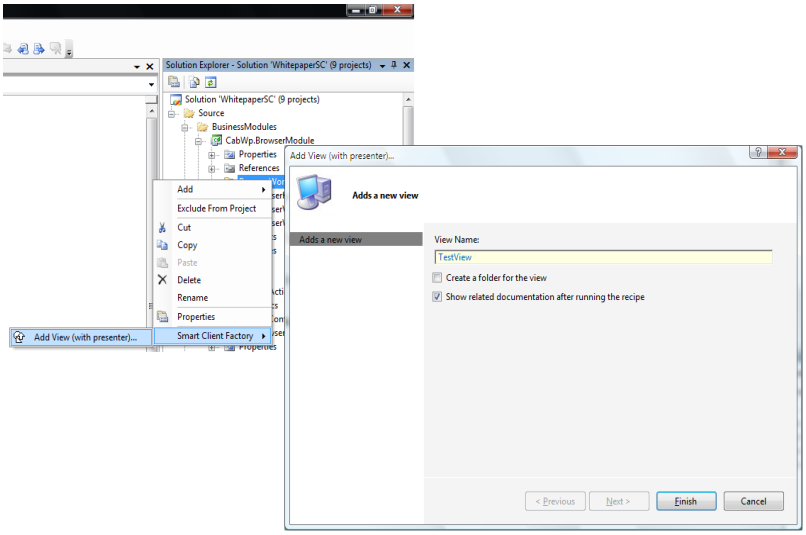


Figure 12: Adding a View with a Presenter

Next you need to instantiate the SmartParts and activate them when the WorkItem gets activated as well. You don't need to take care about the presenter as it gets created automatically 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 via CAB's factory methods. Optionally when calling the AddNew method you can pass in a unique ID for the SmartPart 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 in 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 for launching a WorkItem. Typically you would reserve a command for this purpose which will be invoked through a UIExtensionSite.

Create a Commands for launching first-level-WorkItems

CAB provides a ready-to-use infrastructure for working with the command-pattern. You either can create separate classes with a handler implemented or you can mark methods of your WorkItemController with the [Command] attribute. Checklist 17 outlines the steps for adding a new command to a CAB-based component (WorkItem, Controller, 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 (e.g. 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 event-handlers. Therefore the signature of the command-handler methods adheres to the standard .NET 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 you need to create controls on UIExtensionSites next. As you want to create them based on the user's security (some commands are available to users in some roles, only). SCSF supports so-called action-catalog classes where you can define actions 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 the 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* which implements the IActionCondition interface and is used for validating, if a user is permitted for executing 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* as explain 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 which 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* of and then add the previously added action catalog. Then register the ActionCatalog created in step for 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 example above called all actions immediately 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. E.g. the commands introduced in the section “Create a Commands for launching first-level-WorkItems” implement the logic directly. But you can also create actions for the logic executed in Commands instead of executing the logic directly 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 -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 identifying events uniquely. The checklists below outline the steps for creating an event-publication and -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 implementing the parts of the module. Right-click on the class which should publish the event and use the *Smart Client 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. Decide about the publication-scope as well, keep it as narrow as possible. If an event is just required within the current WorkItem and its sub-WorkItems, then 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 throw 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);
        }
    }
```

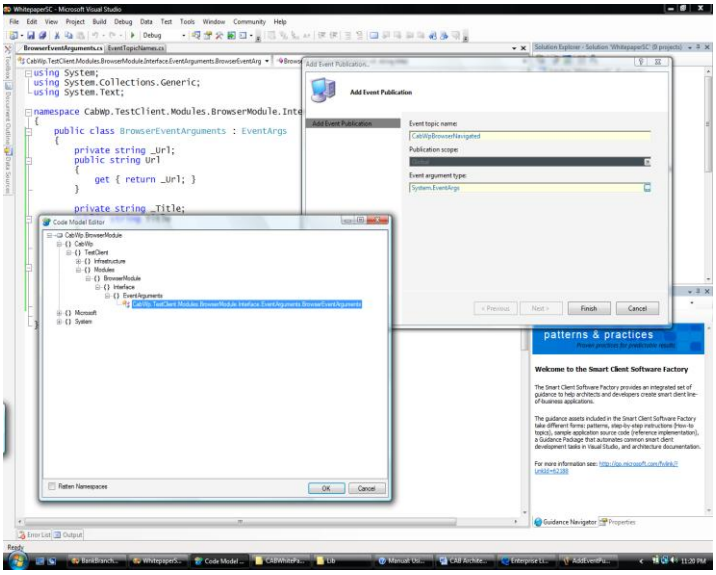


Figure 13: Adding a event-publication using SCSF

Next you can subscribe to the event using the *Smart Client Factory* context menu on the component (WorkItem, Presenter, Controller…) you wish to subscribe to the event as follows.

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 from the context menu select *Smart Client Factory – Add Event Subscription*.
3. In the wizard select the event-subscription from the drop-down 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 wired up with the publisher automatically. In this method you now can add the logic needed to be executed when the subscribed event is received.

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

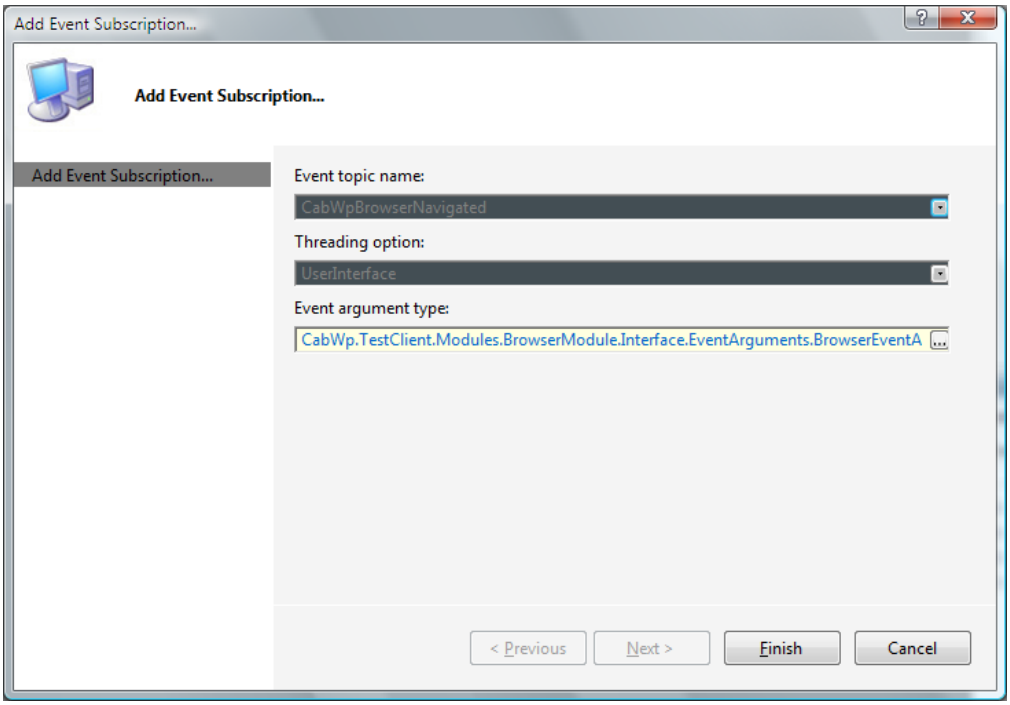


Figure 14: Adding event-subscriptions using SCSF

Configure the Application

Finally you need to configure your application. Primarily there are two parts you need to configure – first the catalog containing the list of modules to be loaded and second 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 on the solution profile refer to the documentation of CAB and SCSF. Next you need to modify the configuration of your app.config – especially for the action-catalog covered in the section “Create an ActionCatalog for creating UIExtensionSites”. Here the steps are the following:

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-provide 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 now you can run and test the application.

Summary

This whitepaper is a guideline 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 on identifying CAB-based components such as WorkItems and services and decide on 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 and an entity-driven strategy for identifying WorkItems and you learned about criteria for packaging WorkItems into modules together.

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

Figure-Index

Checklist-Index