**Computación y Sistemas** Vol. 6 No. 2 pp. 130 - 142
© 2002, CIC - IPN. *ISSN 1405-5546* Impreso en México

# Tabbed Hierarchy: A Design Pattern for the Implementation of Object-Oriented User Interfaces*

***Jerarquía Tabulada: Un Patrón de Diseño para la Implementación de Interfaces de Usuario Orientadas a Objetos***

**Amador Durán, Antonio Ruiz Cortés, Rafael Corchuelo** y **Octavio Martín Díaz**
Departamento de Lenguajes y Sistemas Informáticos
ETS. de Ingeniería Informática, Universidad de Sevilla
Avda. Reina Mercedes s/n 41012 Sevilla, España
E-mail: {amador, aruiz, corchu, octavio}@lsi.us.es

Article received on February 12, 2002; accepted on November 05, 2002

## Abstract

*During the development of object–oriented user interfaces, it is usually necessary to build dialogs for the edition of object properties. More often than not, these dialogs have to layout many user interface elements, especially when object classes are part of inheritance hierarchies and have many properties. In such situations, tabbed dialogs are a good alternative to scrolling. The user interface design pattern presented in this article uses class hierarchies for organizing user interface elements in tabbed dialogs for the edition of object properties. It also promotes reuse of code and user interface resources and a coherent and easy–to–learn user interface style. Results of the application of some refactorings and of the introduced pattern in a case study are also presented.*

**Keywords**: Design Patterns, User Interface Patterns, Refactoring, C++, MFC

## Resumen

*Durante el desarrollo de interfaces de usuario orientadas a objetos, suele ser necesario construir diálogos para editar propiedades de objetos. A menudo, estos diálogos tiene que visualizar muchos elementos de interfaz de usuario, especialmente cuando las clases de los objetos forman parte de jerarquías de herencia y tienen un gran número de propiedades. En estos casos, los diálogos tabulados son una buena alternativa al desplazamiento. El patrón de diseño de interfaz de usuario presentado en este artículo usa las jerarquías de clases para organizar los elementos de interfaz de usuario en diálogos tabulados para la edición de propiedades de objetos. También promueve la reutilización de código y de recursos de interfaz de usuario y un estilo coherente y fácil de aprender. En este artículo también se presenta un caso de estudio con los resultados de la aplicación de varios refactorings y del patrón descrito.*

**Palabras clave**: Patrones de Diseño, Patrones de Interfaz de Usuario, *Refactoring*, C++, MFC

*This work is partially funded by the CICYT project TIC 2000–1106–C02–01 (GEOZOCO).

## 1 Introduction

Sometimes in object–oriented information systems developments, it is necessary to build user interface (UI) dialogs for the edition of objects of classes with many inherited properties. For example, instances of $\mathsf{ClassToBeEdited}_1$ in the class diagram in figure 1 need a dialog with at least 18 UI elements, also known as *widgets* or *controls*, in order to let end users edit their properties. Such a dialog will probably be too big to fit in a single screen and too complex for both developers and users.

What is more, if another leaf class had to be added to the hierarchy in figure 1, for example the $\mathsf{ClassToBeEdited}_2$ class, another very similar, big dialog would have to be built. Although it depends on the UI class library used, more often than not reuse of code and UI resources would be very difficult, apart from cut and paste reuse. In a scenario like this, two questions arise:

- How to solve the problem of too big dialogs for the edition of objects with many, possibly inherited, properties?
- How to reuse common code and common UI resources of very similar dialogs for such situations?

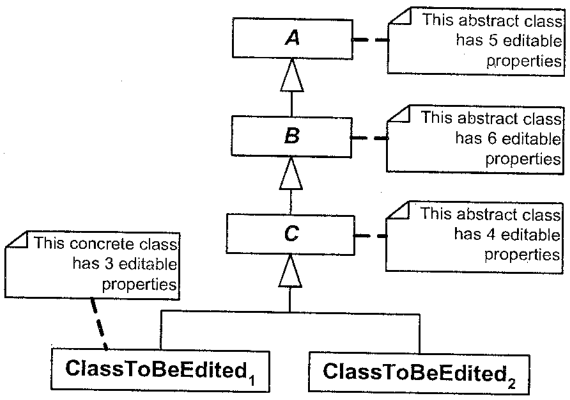

A
This abstract class has 5 editable properties
B
This abstract class has 6 editable properties
C
This abstract class has 4 editable properties
This concrete class has 3 editable properties
ClassToBeEdited1
ClassToBeEdited2

Figure 1. Sample class hierarchy

In this article, we introduce the *Tabbed Hierarchy* design pattern, a design pattern that tries to answer the former questions. This pattern drives the design of tabbed dialogs for the edition of complex objects and it also helps to reuse code and UI resources.

We identified this design pattern during the development of REM (Durán, 2000), an experimental requirements management CASE tool built with Microsoft Visual C++. That is the reason why all the code examples in this article are written using the Microsoft Foundation Library (MFC) classes (Prosise, 1999; Kruglinski *et al.*, 1998). Nevertheless, the resulting code should be clear and easy to read for any average C++ or Java programmer. We have followed most usual MFC naming conventions in our code. For example, all class names start with an upper case C, all property names start with a lower case M and an underscore, as an abbreviation of (data) member, and all pointer attributes or variables start their names with a lower case P.

The rest of the article is organized as follows. In section 2, we introduce some basic concepts about programming UI dialogs with the MFC, which are necessary in order to understand the rest of the article. In section 3, some *refactorings* (Fowler, 1999) are applied to the original MFC code in order to make it easier to use and to promote encapsulation. In section 4, the Tabbed Hierarchy pattern is described using the format proposed in (Gamma *et al.*, 1995). The related work is compared in section 5 and, finally, in section 6, some conclusions are presented.

## 2 User Interface Dialogs in the MFC library

The three main UI dialog classes in the MFC library are: CDialog, for managing ordinary dialog boxes like the one in figure 2; CPropertyPage, for managing dialog boxes inside a tabbed dialog, like the dialogs tabbed as Page 1, Page 2 and Page 3 in figure 3; and CPropertySheet, for managing tabbed dialog containers like the one in figure 3.

In the rest of the article and for the sake of clarity, we will adopt MFC terminology and will use the term *dialog box* for ordinary dialogs, *property page* for dialog boxes inside a tabbed dialog, and *property sheets* for tabbed dialogs containing property pages. For referring to UI elements like buttons, text fields, lists, etc. we will use the term *control*.

### 2.1 Building dialogs with Visual C++

The process of building dialogs with Visual C++ is relatively simple. At first, the developer sets the dialog appearance with the dialog editor of *Visual Studio*, the integrated development environment (IDE) of Visual C++. Then, the Visual Studio integrated code generator, *ClassWizard*, generates a new class, derived from CDialog or from CPropertyPage, associated with the new dialog resource.[1] Once the new class has been generated, the developer can add public data attributes to the new class and associate them with controls in the dialog using the facilities provided by ClassWizard.

When the dialog is displayed, any change in a control will also take effect in its associated attribute and vice versa. The relationships between controls and attributes of the dialog class are made explicit in the virtual DoDataExchange method by means of the so-called DDX (dialog data exchange) and DDV (dialog data validation) functions. A comprehensive discussion of this topic can be found in (Prosise, 1999) or (Kruglinski *et al.*, 1998).

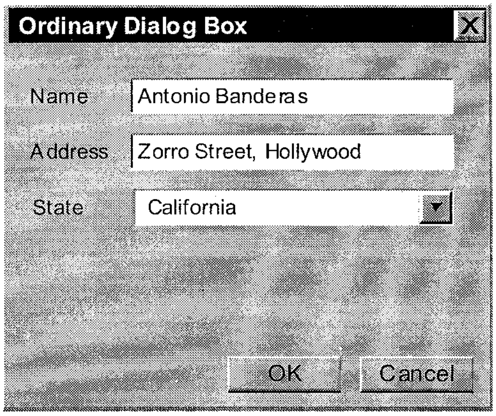


Figure 2. Typical dialog box

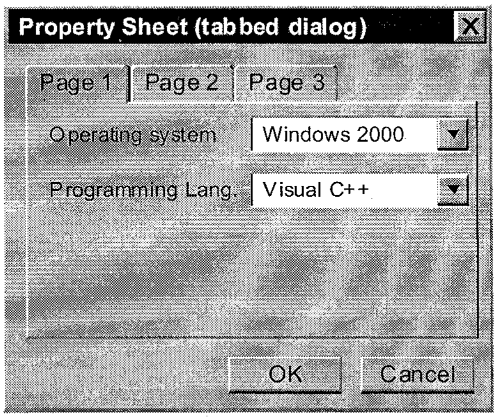


Figure 3. Typical property sheets

[1]In Microsoft Windows, UI resources like dialogs, menus, icons, etc. are defined in so-called *resource files*. These external files have their own syntax and can be used from any programming language. See (Prosise, 1999; Kruglinski *et al.*, 1998) for more details.

## 2.2 Programming object–oriented dialogs with MFC

The usual way of programming object–oriented dialog boxes in Visual C++ consists of the following sequence of steps (see figure 4):

1. Create the dialog box object.
2. Initialize dialog box object attributes with properties from the object to be edited, which we will call the *subject* from now on.
3. Show the dialog box using the inherited DoModal method.
4. Update the subject state if the user pressed the OK button or do nothing in any other case.

Property sheets are used in a similar way. The programmer has to:

1. Create the property sheet object.
2. Create the property page objects.
3. Initialize attributes of property page objects with properties from the subject.
4. Add property page objects to the property sheet object.
5. Show the property sheet using also the inherited method DoModal.
6. Update the subject state with attribute values from property page objects if the user pressed the OK button of the property sheet or do nothing in any other case.

The code in figure 5 shows a typical usage of a property sheet. Notice that the CPropertySheet class is used directly; there is no need for derivation.

## 3 Refactoring the MFC code

Refactoring is a technique to restructure code in a disciplined way which has become popular after the publication of (Fowler, 1999). The concept of refactoring is defined in (Fowler, 1999) as "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*". In other words, refactorings *clean up* code in a controlled way, thus minimizing the introduction of new bugs.

As pointed out by Gamma in the foreword in (Fowler, 1999), "*design patterns provide targets for refactorings*". So, before presenting the Tabbed Hierarchy design pattern, some *pattern–driven* refactorings of the original MFC code are needed. These refactorings will increase encapsulation and will ease the application of the Tabbed Hierarchy pattern.

In order to make this article as self–contained as possible, a brief description of the main principles behind the refactorings to be applied is provided as follows:

**Extract Method**
*If you have a code fragment that can be grouped together, turn the fragment into a method whose name explains the purpose of the method.*

**Preserve Whole Object**
*If you are getting several values from an object and passing these values as parameters in a method call, send the whole object instead.*

**Remove Parameter**
*If a parameter is no longer used by the method body, remove it.*

**Encapsulate Field**
*If there is a public field (i.e. attribute, property), make it private and provide accessors.*

**Pull Up Method**
*If you have methods with identical results on subclasses, move them to the superclass.*

### 3.1 Encapsulating dialog boxes and property pages

In the usual MFC code for programming object–oriented dialog boxes and property sheets (see figures 4 and 5), two main blocks can be identified: *dialog initialization* (everything after dialog creation and before dialog activation) and *subject update* (everything inside the body of the if statement). This usual MFC code presents three main problems:

1. Every time the dialog is used, all the code for assignments needed for dialog initialization have to be repeated.
2. All the updating messages sent to the subject have also to be repeated every time the dialog is used.
3. Dialog attributes are public, which is not considered as a good practice because it violates the encapsulation principle.

With respect to the first problem, the initialization fragment could be turned into a method applying the Extract Method refactoring. Since this method would be invoked immediately after dialog creation, its code could be allocated into dialog constructor instead of in its own method. All parameters needed by the new constructor would be obtained from the subject, so it could be passed as a parameter, following the Preserve Whole Object refactoring.

```
CMyDialogBoxDlg dlg; // (1) dialog object creation
dlg.m_x = subject.GetX(); // (2) dialog object initialization
dlg.m_y = subject.GetY(); // (2) dialog object initialization
...
if ( dlg.DoModal() == IDOK ) { // (3) dialog box activation
    subject.SetX( dlg.m_x ); // (4) subject update
    subject.SetY( dlg.m_y ); // (4) subject update
    ...
}
```

Figure 4. Usual MFC code for object–oriented dialog boxes

```
CPropertySheet sheet; // (1) property sheet creation
CMyPage1 page1;                // (2) property page creation
page_1.m_x = subject.GetX(); // (3) property page initialization
...
CMyPage2 page2;                // (2) property page creation
page_2.m_y = subject.GetY(); // (3) property page initialization
...
sheet.AddPage( &page1 ); // (4) association of pages to the sheet
sheet.AddPage( &page2 ); // (4) association of pages to the sheet
...
if ( sheet.DoModal() == IDOK ) { // (5) property sheet activation
    subject.SetX( page_1.m_x ); // (6) subject update
    ...
    subject.SetY( page_2.m_y ); // (6) subject update
    ...
}
```

Figure 5. Usual MFC code for object–oriented property sheets

```
CMyDialogBoxDlg dlg( &subject ); // (1) dialog creation/initialization

if ( dlg.DoModal() == IDOK ) // (2) dialog activation
    dlg.UpdateSubject();     // (3) subject update
```

Figure 6. Refactored code for dialog box programming

```
CPropertySheet sheet; // (1) property sheet creation
CMyPage1 page1( &subject ); // (2) property page creation/initialization
CMyPage2 page2( &subject ); // (2) property page creation/initialization
...
sheet.AddPage( &page1 ); // (3) association of pages to sheet
sheet.AddPage( &page2 ); // (3) association of pages to sheet
...
if ( sheet.DoModal() == IDOK ) { // (4) property sheet activation
    page_1.UpdateSubject(); // (5) subject update
    page_2.UpdateSubject(); // (5) subject update
    ...
}
```

Figure 7. Refactored code for property sheet programming

In order to solve the second problem, the Extract Method refactoring can also be applied. In this way, the subject update code fragment is refactored into a new method, UpdateSubject, which would be responsible for sending the messages needed for updating the subject state. If the Extract Method and the Preserve Whole Object refactorings were strictly applied to this code fragment, the UpdateSubject method would have to take the subject as a parameter. Since this subject is the same object that was passed as a parameter to the constructor, it seems to be a better choice to store a pointer to the subject inside the dialog object, thus avoiding the need for a parameter in UpdateSubject. This could be considered as a variant of the Remove Parameter refactoring.

Applying the proposed refactorings, the resulting code would be as shown in figures 6 and 7. As it can be seen, dialog attributes need not be public anymore, so the Encapsulate Field refactoring can also be applied, thus solving the third problem. Notice that no public accessors for attributes were added since there was no need for accessing them from outside the dialog class code.

## 3.2 Encapsulating property sheets

Refactorings applied in previous section have dramatically simplified the code for dialog programming and have promoted encapsulation, but in the case of property sheets some potentially *extractable* fragments of code are still present (steps 2, 3 and 5 in the listing in figure 7). These code fragments must be repeated every time a property sheet is used, so they should be allocated into their own method applying the Extract Method refactoring once again.

Since extracted methods would have different code, we need to create new subclasses of CPropertySheet. Sample code for these new subclasses is shown in figure 8, where property pages have been turned into embedded objects.

Using these new classes of property sheet, the final code would be as shown in figure 9. As it can be seen, the resulting code is as simple as the code for dialog boxes and it has the same programming interface, since the UpdateSubject method has also been added to the new property sheet classes. What is more, the client code does not depend anymore on the specific property pages used inside the property sheet code.

## 3.3 The UpdateSubject protocol

If many property sheets are going to be used like the previously shown, some common code could be allocated into an abstract class applying the Pull Up Method refactoring. In fact, the common code is a protocol for subject updating between a property sheet and its subordinate property pages. The main steps of this protocol are:

1. The property sheet must propagate the UpdateSubject message towards its property pages.
2. The property pages must implement the UpdateSubject method with code for updating the subject.

The UML class diagram in figure 10 shows the basic structure. The CTHPropertySheet class (TH stands for *Tabbed Hierarchy*) inherits from the MFC class CPropertySheet and declares a single public method, UpdateSubject. In this method, the property sheet iterates over its inherited array of property pages sending the UpdateSubject message to all of them implementing the CTHPropertyPage interface (see figure 11). To test if a property page implements the CTHPropertyPage interface, we use the C++ keyword dynamic_cast, that returns a null pointer if the casting is not possible.

The CTHPropertyPage class is just an interface, what in C++ means it only has pure virtual methods. Any derived property page class must implement the only pure virtual method declared, UpdateSubject, which is responsible for updating subject state.

## 4 The Tabbed Hierarchy Design Pattern

Having introduced some basic concepts and refactorings about dialog programming with the MFC, we will now present the Tabbed Hierarchy (TH) design pattern. For the pattern description, we will use a simplified version of the template used in the pattern catalog in (Gamma *et al.*, 1995).

### 4.1 Intent

The intent of the TH design pattern is building reusable tabbed dialogs (property sheets) for object–oriented UIs.

The basic concept behind the TH pattern is shown in figure 12, where the hierarchy of classes in figure 1 is shown in a *tabbed* form. If a property sheet for the edition of objects of a given leaf class is needed, build at least one property page per class in the hierarchy, so common superclasses will have common property pages which will be reused in different property sheets. For example, property pages for A, B and C classes would be reused in the property sheets for $\text{ClassToBeEdited}_1$ and $\text{ClassToBeEdited}_2$ classes, as suggested in figure 12.

### 4.2 Motivation

During the development of the requirements management CASE tool REM (Durán, 2000), we identified the class hierarchy shown in figure 13, with 3 levels of inheritance, 3 abstract classes and 21 leaf classes. We had to develop one edition dialog for each leaf class. We chose property sheets because they seemed to be a better choice than scrolling for organizing dialogs with many controls. When we had to make a decision about what controls were in each property

```
class CMySheet  : public CPropertySheet {
private:
   CMyPage1 m_page1;          // (1) property page declaration
   CMyPage2 m_page2;          // (1) property page declaration
public:
   CMySheet( CMyClass* pSubject )
   : m_page1( pSubject ),     // (2) property page initialization
     m_page2( pSubject )      // (2) property page initialization
   {
      AddPage( &m_page1 );    // (3) association of pages to sheet
      AddPage( &m_page2 );    // (3) association of pages to sheet
   };
   void UpdateSubject()
   {
      m_page1.UpdateSubject(); // (4) subject update
      m_page2.UpdateSubject(); // (4) subject update
   };
};
```

Figure 8. Sample code for refactored property sheet class

```
CMySheet sheet( &subject ); // (1) property sheet creation/initialization

if ( sheet.DoModal() == IDOK ) // (2) property sheet activation
    sheet.UpdateSubject();     // (3) subject update
```

Figure 9. Usage of refactored property sheets

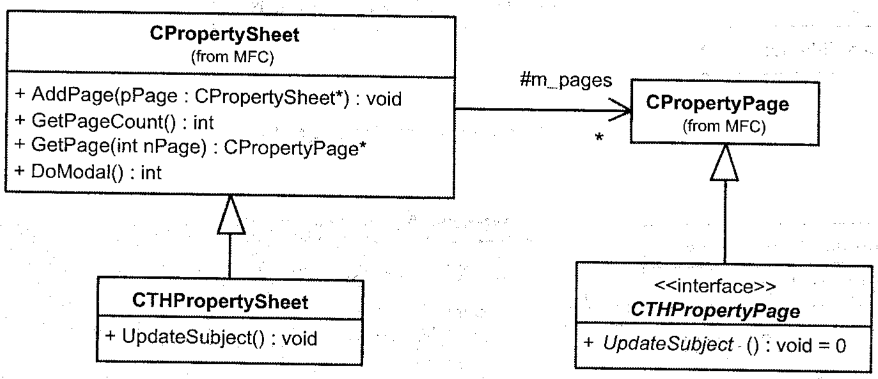


Figure 10. Base classes for the UpdateSubject protocol

```
void CTHPropertySheet::UpdateSubject()
{
    for ( int i = 0; i < GetPageCount(); i++ ) {
       CTHPropertyPage* pPage = dynamic_cast< CTHPropertyPage* >( GetPage( i ) );
       if ( pPage != NULL )
          pPage->UpdateSubject();
    }
}
```

Figure 11. UpdateSubject method

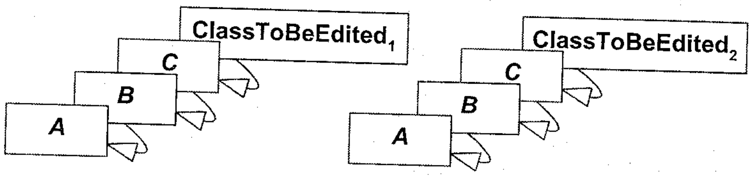


Figure 12. Tabbed hierarchies

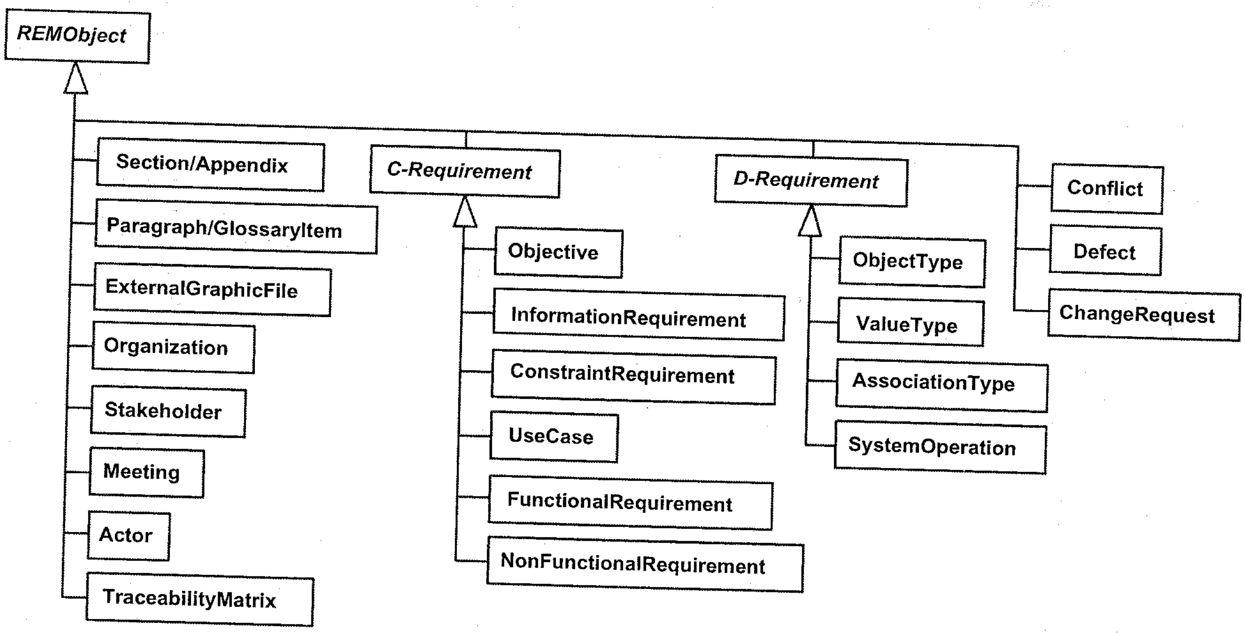
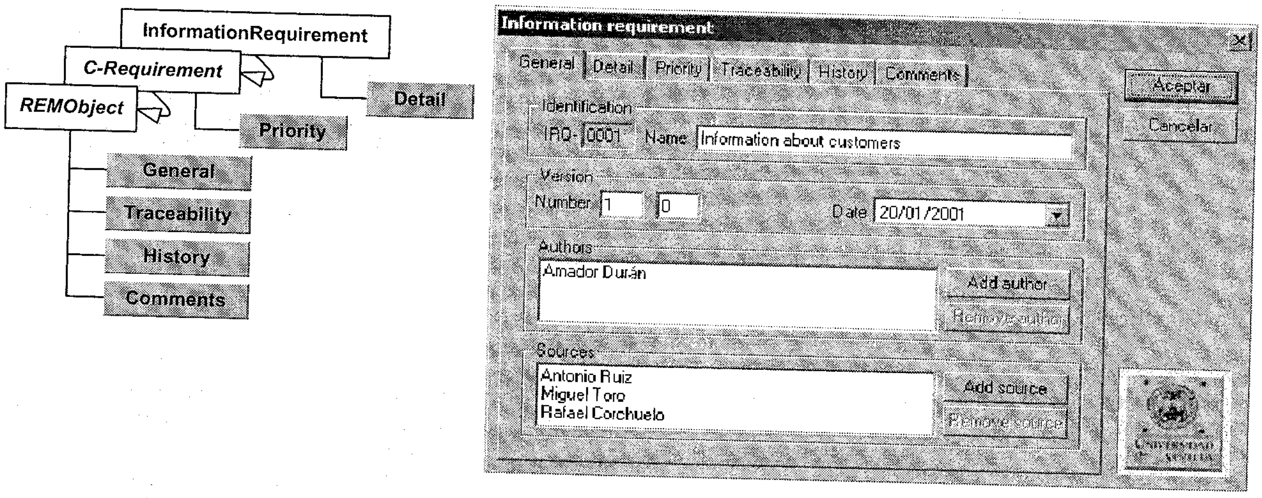


Figure 13. REM class hierarchy



Identification
IRQ- 0001 Name Information about customers
Version
Number 1 0 Date 20/01/2001
Authors
Amador Durán
Sources
Antonio Ruiz
Miguel Toro
Rafael Corchuelo

Figure 14. Property sheet for a REM information requirement

page, we chose to organize property pages imitating the class hierarchy in figure 13.

For example, the property sheet for the edition of information requirements can be seen in figure 14. Property pages labeled as General, Traceability, History and Comments are common for all REM objects. The Priority tab makes sense only for customer requirements (C-Requirements in the hierarchy in figure 13). Specific properties about information requirements are inside the Detail property page, which is the only property page specially developed for this class of objects. The other property pages are reused all along the class hierarchy in figure 13.

## 4.3 Applicability

The TH pattern can be applied in order to build reusable tabbed dialogs for the edition of objects of classes with many common inherited properties.

## 4.4 Structure

In figure 15, the static structure of the TH pattern is shown. Hot spots of the pattern (Pree, 1995) are highlighted following the style used in (Lucena and Fontoura, 2001), and the subject hierarchy has been limited to two levels for simplicity (CAbstractSubject and CConcreteSubject classes respectively). The dynamic structure of the pattern is shown in figure 16, using a UML object diagram.

## 4.5 Participants

- **CTHPropertySheet**: this class inherits from CPropertySheet and declares and implements the UpdateSubject method. It is the base class for all new property sheets using the TH pattern.
- **CTHPropertyPage:** this interface inherits from CPropertyPage and simply declares the UpdateSubject pure virtual method. Any property page using the TH pattern must inherit from this class and therefore implement the UpdateSubject abstract method.
- **CAbstractSubject:** it is the root class of the hierarchy of subjects.
- **CConcreteSubject**: any leaf class in the hierarchy of subjects.
- **CConcreteSubjectSheet:** a concrete class inheriting from CTHPropertySheet. It has as many object members as property pages are needed, usually one page for every class in the subject hierarchy, although it is also possible to have more than one page per class, as in the motivation example. It receives a pointer to the subject in its constructor and passes it to its embedded property pages in its initialization list.
- **CAbstractSubjectPage:** a property page class especially designed for objects of the CAbstractSubject class. It sees the subject as an instance of the CAbstractSubject class, regardless of its dynamic class. It implements the UpdateSubject method and stores a private pointer to the subject (m_pSubject).
- **CConcreteSubjectPage:** a property page class especially designed for objects of the CConcreteSubject class. It lets the user edit the non-inherited properties of CConcreteSubject class. It also implements the UpdateSubject method and stores another private pointer to the subject.

## 4.6 Collaborations

- When the concrete property sheet is created, it receives a pointer to the subject and passes it to its embedded property pages. Property pages store the pointer to the subject and query the subject state as needed for the initialization of their control–associated attributes (see top of figure 17).
- When the property sheet receives the UpdateSubject message, it iterates over its property pages propagating the message. Property pages update subject state inside their UpdateSubject methods (see bottom of figure 17).

## 4.7 Consequences

This pattern has the benefits of organizing tabbed dialogs in a homogeneous way for the user and promoting reuse. The user perceives the commonality between property sheets, so the learning time is shorter. When used with MFC, the only drawback is that code automatically generated by the Visual C++ IDE has to be refactored manually.

## 4.8 Implementation

Using templates should be considered in situations in which many sibling subclasses exist at the bottom level of the subject hierarchy and they differ only in the type of specific property page. The code for a template property sheet is shown in figure 18, where the formal parameters have been underlined.

## 4.9 Sample code

Including a whole example would take too much space, so we have taken two fragments directly from the code of our requirements management CASE tool REM (see section 4.10 for more details about REM).

The fragment of code in figure 19 corresponds to the property sheet class for use cases, a subclass of customer requirements that are in turn a subclass of REM objects (see figure

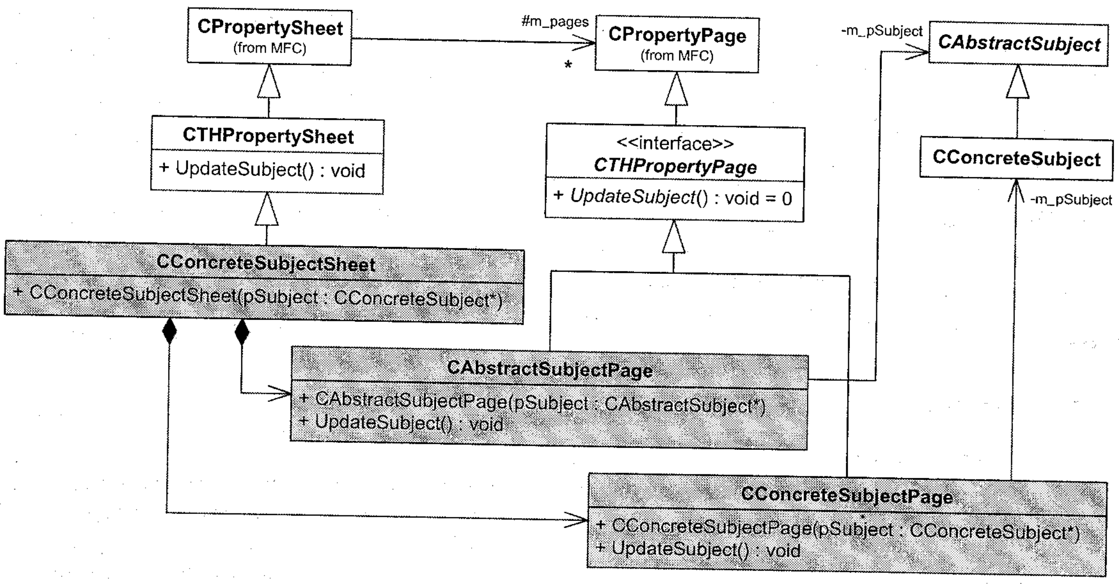


Figure 15. Tabbed hierarchy pattern static structure

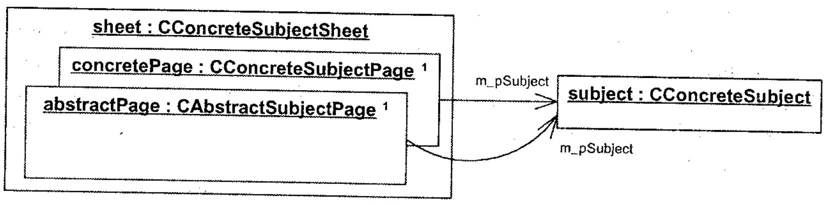

sheet : CConcreteSubjectSheet
concretePage : CConcreteSubjectPage 1
abstractPage : CAbstractSubjectPage 1
m_pSubject
subject : CConcreteSubject
m_pSubject

Figure 16. Tabbed hierarchy pattern dynamic structure

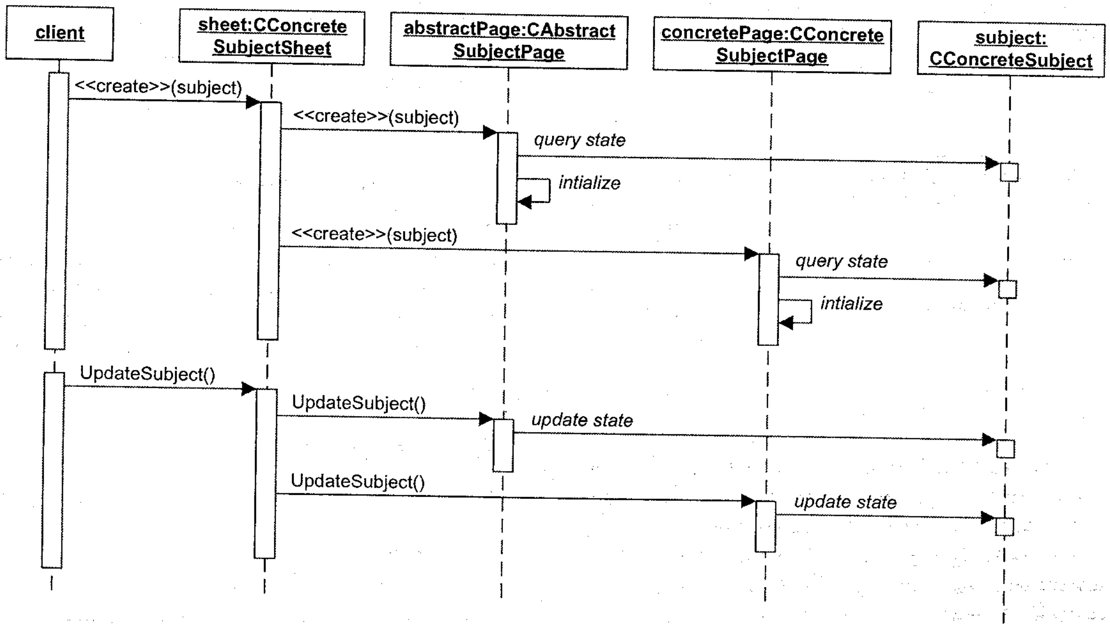


Figure 17. Tabbed hierarchy pattern collaborations

```
template <class LeafClass, class LeafPage>
class CTemplateSheet : public CTHPropertySheet {
private:
   // declaration of common pages
   LeafPage m_leafPage; // specific page

public:
   CTemplateSheet( LeafClass* pSubject ) :
   // initialization of common pages
   m_leafPage( pSubject )
   {
      // addition of common pages
      AddPage( &m_leafPage );
   };
};
```

Figure 18. Template version of CTHPropertySheet

```
class CUseCaseDlg : public CREMSheet {
public:
CUseCaseDlg( CUseCase* pSubject, UINT nIDCaption = IDS_REM_USE_CASE )
: CREMSheet( nIDCaption ),
  m_generalPage( pSubject ),
  m_detailPage( pSubject ),
  m_prePostPage( pSubject ),
  m_priorityPage( pSubject ),
  m_traceabilityPage( pSubject ),
  m_commentsPage( pSubject )
{
    AddPage( &m_generalPage      );
    AddPage( &m_detailPage       );
    AddPage( &m_prePostPage      );
    AddPage( &m_priorityPage     );
    AddPage( &m_traceabilityPage );
    AddPage( &m_commentsPage     );
};

private:
    CREMObjectPage       m_generalPage;
    CUseCasePage         m_detailPage;
    CPrePostPage         m_prePostPage;
    CC_RequirementPage   m_priorityPage;
    CTraceabilityPage    m_traceabilityPage;
    CCommentsPage        m_commentsPage;
};
```

Figure 19. Property sheet class for REM use case objects

```
void CInformationRequirementPage::UpdateSubject() {
    m_pSubject->SetRelevantConcept( m_relevantConcept );
    m_pSubject->SetAvgLifeTimeValue( m_avgLifeTimeValue );
    m_pSubject->SetAvgLifeTimeTime( m_pAvgLifeTimeTime );
    m_pSubject->SetMaxLifeTimeValue( m_maxLifeTimeValue );
    m_pSubject->SetMaxLifeTimeTime( m_pMaxLifeTimeTime );
    m_pSubject->SetAvgOcurrences( m_avgOcurrences );
    m_pSubject->SetMaxOcurrences( m_maxOcurrences );
}
```

Figure 20. UpdateSubject code for REM information requirement objects

13). As other property sheets classes in REM, this property sheet class derives from CREMSheet, an abstract class derived from CTHPropertySheet, which takes a string identifier for dialog title as a parameter in its constructor.

As it can be seen, the code is quite simple. The property sheet for use cases declares 6 property pages, but only 2, CUseCasePage and CPrePostPage, are specific to the class. All used property pages derive from CTHPropertyPage and therefore implement the UpdateSubject method.

The fragment of code in figure 20 corresponds to the code for the UpdateSubject method of the CInformationRequirementPage class, where some properties of information requirements (also a subclass of customer requirements) are updated.

## 4.10 Known uses

As we have already mentioned, we have used this pattern in our requirements management CASE tool REM (Durán, 2000). During the development of REM, we identified 21 leaf classes in a hierarchy of 3 levels (see figure 13). The number of property pages needed for the 21 leaf classes was 119, but only 33 had to be different after applying the TH pattern. In other words, the reuse level of UI resources was of 72%.

Applying the template version of the pattern (see figure 18), we developed 3 template property sheets (one for each abstract class), which were reused for 14 out of the 21 leaf classes. Only 7 property sheets had to be written entirely. The main reason for writing specific property sheets was that some leaf classes needed more than one property page, so it was not possible to write a generic property sheet. Anyway, the reuse level of UI dialog code was almost as high as for UI resources.

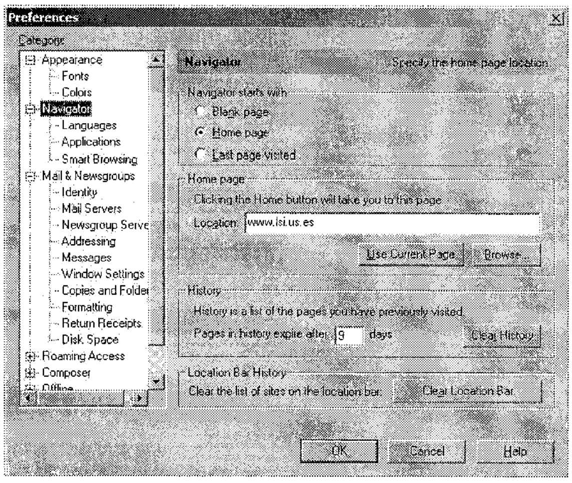
## 4.11 Related patterns

A simplified version of *Chain of Responsibility* (Gamma *et al.*, 1995) is used between the property sheet and its property pages for the implementation of the UpdateSubject protocol.

The *Navigating between Spaces* interaction pattern (van Welie and Troetteberg, 2000) suggests that "*when the user needs to access an amount of information which cannot be put on the available space*", the best solution is "*showing the information in several spaces and allow the user to navigate between them*". In the TH pattern, the several spaces are the property pages and the user can navigate between them using the tabs.

# 5 Related Work

General concepts about object–oriented user interfaces are presented in (Collins, 1995). More specifically, we must refer to the excellent catalog of interaction patterns (van Welie and Troetteberg, 2000) presented in the 2000 edition of the *Patterns Languages of Programming* (PLoP) conference. The TH design pattern can be considered as a more concrete version of one of the interaction patterns described in (van Welie and Troetteberg, 2000), namely *Navigating between Spaces*, as we have already commented in section 4.11.



Navigator starts with: ( ) Blank page [✓ Home page] ( ) Last page visited

Clicking the Home button will take you to this page

Location: www.lsi.us.es

History is a list of the pages you have previously visited.

Pages in history expire after: 9 days

Clear the list of sites on the location bar

Figure 21. Tree-based dialog

There is also an interesting work on UI *antipatterns*[2] known as *The Interface Hall of Shame* (IIS, 2000). In that catalog of antipatterns, there is a section especially focused on tabbed dialogs. Many of the reported problems of tabbed dialogs have to do with using too many tabs or with an inconsistent grouping of controls under different tabs. The TH design pattern avoids most of those problems.

In both (van Welie and Troetteberg, 2000) and (IIS, 2000), there is also a clear *subpattern*: "*if the number of spaces (tabs) is large, for example greater than 8, use a tree structure instead of tabbed dialogs*". In figure 21, from Netscape Navigator, an application of that subpattern can be seen. Dialog appearance changes when the user selects a different item in the tree on the left, which is a better option than using more than 30 tabs which would be needed in case of using a tabbed structure.

Although it has not been the case during the development of REM, it is clear that the TH design pattern is meant to be used with not very deep class hierarchies, *i.e.* no more than 6 or 7 levels of inheritance, in order to keep the number of tabs reasonable. Since such deep hierarchies are not found very often in domain models, for example the one shown in figure 13, the TH pattern can be applied in most developments. If the number of tabs had to be larger, the solution proposed in (van Welie and Troetteberg, 2000) and (IIS, 2000) of using

[2]Antipatterns describe solutions to problems that generate negative consequences, their causes, symptoms, and refactored solutions (Brown *et al.*, 1998).

a tree for dialog selection could be adopted, but the pattern structure should then be changed.

## 6 Conclusions

In this article, the Tabbed Hierarchy user interface design pattern has been presented. This pattern drives the design and implementation of tabbed dialogs for object–oriented user interfaces, achieving a high level of reuse. It has been applied during the development of REM, an experimental requirements management tool recently presented at the research tool demo session of the *IEEE Joint International Requirements Engineering Conference* held in Essen (Germany) on September 2002, with excellent results in UI homogeneity and in code and UI resources reuse. The reader can check the application of the presented pattern downloading REM from `http://klendathu.lsi.us.es/REM`.

In spite of the fact that the code presented in this article is obviously MFC–oriented, an adaptation to the *Swing* library, a package in the Java Foundation Classes (JFC) suite, is not complex (Eckstein *et al.*, 1998). Although there is no a direct mapping, only a few changes have to be applied using a JOptionDialog with an embedded JTabbedPane as a substitute of CPropertySheet, and JPanel instead of CPropertyPage.

## References

**Booch, G.**, **Rumbaugh, J.**, and **Jacobson, I.**, *The Unified Modeling Language User Guide*, Addison–Wesley, 1999.

**Brown, W. J.**, **Malveau, R. C.**, **McCormick III, H. W.**, and **Mowbray, T. J.**, *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*, Wiley, 1998.

**Collins, D.**, *Designing Object–Oriented User Interfaces*, Benjamin/Cummings, 1995.

**Durán, A.**, "A Methodological Framework for Requirements Engineering of Information Systems" (in Spanish), PhD thesis, University of Seville, 2000.

**Durán, A.**, **Ruiz-Cortés, A.**, **Corchuelo, R.**, and **Toro, M.**, "The Tabbed Hierarchy Desing Pattern", in *V Workshop Iberoamericano de Ingeniería de Requisitos y Desarrollo de Ambientes de Software*, La Habana, Cuba, April, 2002, pp. 417–416.

**Eckstein, R.**, **Loy, M.**, and **Wood, D.**, *Java Swing*, O'Reilly, 1998.

**Fowler, M.**, *Refactoring: Improving the Design of Existing Code*, Addison–Wesley, 1999.

**Gamma, E.**, **Helm, R.**, **Johnson, R.**, and **Vlissides, J.**, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison–Wesley, 1995.

**Isys Information Architecths**, "Interface Hall of Shame", 2000. It can be downloaded from `http://www.iarchitect.com/mshame.htm`.

**Kruglinski, D. J.**, **Wingo, S.**, and **Shepherd, G.**, *Programming Visual C++*, Microsoft Press, fifth edition, 1999.

**Lucena, C. J. P.** and **Fontoura, M.**, "Extending UML to Improve the Representation of Design Patterns", in *Journal of Object–Oriented Programming*, 13(11), 2001.

**Pree, W.**, *Design Patterns for Object–Oriented Software Development*, Addison–Wesley, 1995.

**Prosise, J.**, *Programming Windows with MFC*, Microsoft Press, second edition, 1999.

**van Welie, M.** and **Troetteberg, H.**, "Interaction Patterns in User Interfaces", in *Proceedings of the 7th Pattern Languages of Programming Conference*, 2000.

***Amador Durán,*** *was born in 1970, in Seville, Spain. He received his MS degree in Computer Science in 1993 and his Ph.D. in Computer Science in 2000, both from the University of Seville. He is an assistant professor at the Department of Computer Languages and Systems of the University of Seville since 1994. His current research interests are Requirements Engineering, Software Development Process Modeling, Conceptual Modeling, Software Engineering for Web Applications and Software Engineering Patterns at any level of abstraction.*

***Antonio Ruiz Cortés,*** *is a Doctor of Computer Science, and he is with the Department of Computer Languages and Systems of the University of Seville. Before joining the University, he worked for companies such as Informática el Corte Inglés, which is a leading Spanish corporation with branches all over the world, and DEINSA, which is a local corporation that has specialised in building environmental information systems. His current research interests include non—functional requirements, software architecture, component—oriented software engineering, and multi—organizational web—based systems.*

***Rafael Corchuelo,*** *is a reader in Computer Engineering, and he has been with the University of Seville since 1994. He is the head of the Research Group on Distributed Systems of this University, and he has set up several cooperation and exchange programmes with several European universities and research centers. His research activities focus on distributed systems. Currently, he is a member of the editorial board of Springer—Verlag's Journal of Universal Computer Science, and serves as a reviewer for ACM's Computing Reviews and Wiley's Concurrency and Computation.*

***Octavio Martín Díaz,*** *is a lecturer professor of software engineering and a member of the Department of Computer Languages and Systems of the University of Seville since 1998. He received his MS degree in Computer Science from the University of Seville in 1994. He has worked as a software consultant for his County Council where he participated in projects on large database applications. He is currently very interested in software architecture and his research is related to architectural analysis and transformation.*