Appendix C

Microsoft Foundation Classes Pattern Language

A Generative Pattern Language for Writing Applications Using the Microsoft Foundation Classes

Pattern # 1: Create Application Context

Problem:

The framework provides the basic logic for the application, but your problem has special characteristics

Context:

Writing a Windows-based application using the Visual C++ MFC framework.

Forces:

Each application is different in some ways
Each Windows application is the same in many ways including the basic control logic.

Solution:

Create a subclass of **CWinApp** and provide the application-specific logic.

```
class Brickles : public CWinApp// The Application Class
{
  public :
    virtual BOOL InitInstance ();// Set up the Game

} ;
```

Resulting Context:

The application has a basic main method so that the application can be compiled and executed.

Related Patterns:

This is a high level pattern that is related to several others, but it is directly related to **Create Main FrameWindow.**

Pattern # 2: Create a Main FrameWindow

Problem:

Windows expects that an application is tied to a single window.

Context:

Working within the context of a single document interface (SDI) application.

Forces:

Every Windows application requires a single root window.

Solution:

Create a subclass of **CFrameWnd**. Use the Singleton pattern to guarantee that there is only one instance of the main window.

```
class VBrickles : public CFrameWnd// Main Window class
{
    CStatic *title, *pucksMsg;// Information Windows
    CButton *step;// Control Button
    VField  *field;// Main Playing Field

    CBitmap puckBitMap, paddleBitMap;// Bitmaps for Puck and
Paddle
    CBitmap brickOddBitMap; //    and two different styles
    CBitmap brickEvenBitMap;//    of bricks

  public :
    VBrickles ();// Constructor
    ~VBrickles ();// Destructor

    void LoseMsg ();                    // Called when Player
Loses
    void WinMsg ();// Called when Player Wins
    void ContMsg ();// Ask Player to Continue

    afx_msg void StepButton ();// Executed when STEP is Pressed
    afx_msg void ExitCmd ();// Menu Command functions
    afx_msg void AboutCmd ();
    afx_msg void RulesCmd ();

    DECLARE_MESSAGE_MAP()
} ;
```

Resulting Context:

The application has a main window that provides the context for the SDI application.

Related Patterns:

This pattern leads into the **Maintain Status of Resources** pattern.

Pattern # 3: Organize Information into Documents

Problem:

The application-specific information must be modeled in the system

Context:

The MFC framework provides generic behavior for a Windows application, but it can not provide the behavior required in a specific application. The applicationd eveloper must provide that information.

Forces:

The information in an object-oriented application is already organized into objects.
Objects are too fine-grained to provide the architectural design of an application.

Solution:

The application-specific information is organized into documents using subclasses of the MFC **CDocument** class.

```
class ArcadeGame:public CDocument{
   public:
     ArcadeGame(char *name);
     ArcadeGame();
     ~ArcadeGame();

     ArcadeGameMatch* getMatch();
     virtual void Serialize(CArchive& ar);
     virtual void AssertValid() const;
     virtual void Dump(CDumpContext& dc) const;

     virtual PlayingField* newPlayingField();
      // returns a playing field for playing a
      // match of this arcade game
      // (a factory method)
};
```

Resulting Context:

Each piece of information is associated with a document object.

Related Patterns:

This pattern leads into the **Organize Documents Using Document Templates** and the **Associate Views with Documents** patterns.

Pattern # 4: Organize Documents Using Document Templates

Problem:

There is a need to define the basic types of work products that will be handled by an application.

Context:

Within the application context, the menus, icons and other resources that apply to a specific unit of work can be shared.

Forces:

Each unit of work, such as a drawing being created in a draw program or a paper being written in a word processor, constitutes a type of work product.
The types of work products handled by the application will expand with time.
Different types of work products require different menu entries, different responses to the same menu items and require other resources.
There is much overhead in managing different types of work that an application can perform.

Solution:

For an SDI application, create an instance of **CSingleDocTemplate** for each type of document upon which the application will work. This instance will coordinate documents, views and the windows within which they are presented.

Resulting Context:

Every unit of work, a document, is associated with the document template for that type of work product.

Related Patterns:

This pattern is related to the **Organize Documents Using Document Templates** pattern and the **Associate Views with Documents** pattern.

Pattern # 5: Associate Views with Documents

Problem:

The information contained in a document is more stable and better understood than the different ways in which the information is presented to the user.

Context:

Each document provides a model of some piece of domain information.

Forces:

Separate classes for information and presentations of the information are hard to keep in synchronization.
Separate classes for information and presentations of the information provide the flexibility of modifying or adding different presentations of information without modifying the basic information classes.

Solution:

Use a modified Model-View-Controller pattern in which the document is the model and there are associated views. Each instance of subclasses of **CDocument** has an associated set of instances of subclasses of **CView**. The **CSingleDocTemplate** instance associates view classes with document classes.

Resulting Context:

The model of the domain information and the presentation of that information to the user have been decoupled.

Related Patterns:

This pattern is associated with the **Organize Documents Using Document Templates** pattern.

Pattern # 6: Map Windows Messages to Your Application

Problem:

There must be an asociation between each Windows message and themethods of the object that must handle the message within your application.

Context:

Windows generates messages that represent important events, in terms of the user and the operating environment, to which the application must react. These messages are asociated with different methods in the application depending upon which object handles the incoming message.

Forces:

Desire to maintain portability among operating systems
Desire to have context sensitive reaction to external events.

Solution:

Use Message Map macros to associate Windows messages with specific methods in **CCmd-Target** subclasses that will handle the messages.

```
BEGIN_MESSAGE_MAP(VBrickles, CFrameWnd)// Responses to
   ON_BN_CLICKED(IDB_STEPBUTTON, StepButton)
   ON_COMMAND(IDD_EXIT, ExitCmd)
   ON_COMMAND(IDD_ABOUT, AboutCmd)
   ON_COMMAND(IDD_RULES, RulesCmd)
END_MESSAGE_MAP()
```

Resulting Context:

Responsibility for handling a message has been assigned to the method that has the appropri-ate behavior to handle the message.

Related Patterns:

This pattern leads into the **Associate user interface logic with application logic** pattern.

Pattern # 7: Maintain Status of Resources

Problem:

There is a need to maintain a database of information about resources such as pallettes, pens and brushes.

Context:

A program with a graphical interface utilizes many resources such as different fonts, line styles, join types, etc. These resources must be organized to be used effectively.

Forces:

There are a large number of attributes used to control activities such as drawing or writing text.
The user should not be required to reset attributes that really should stay the same most of the time.
These attributes should be grouped logically.

Solution:

Create instances of the **CDC** class or one of its subclasses to manage the contexts for pens, brushes, pallettes and similar resources.

Resulting Context:

The attributes used for a specific context can be shared among several resources or duplicated in other windows to provide uniformity.

Related Patterns:

This pattern is related to the **Create Main FrameWindow** pattern.

Pattern # 8: Associate User Interface Logic with Application Logic

Problem:

There must be an asociation between the Windows messaging architecture an d the object messaging architecture within your application.

Context:

Message maps allow a class to associate an incoming Windows message with a method, but not all messages can be handled by any particular class.

Forces:

We wish to remain as independent of a specific operating system as possible.
The set of objects available to handle messages will change dynamically.

Solution:

Derive message handling classes from subclasses of **CCmdTarget**, such as **CView**, **CDocument**, and **CSingleDocTemplate**. Each instance determines whether it can handle a message or whether it should route the message on to another framework class.

```
class BricksView : public CView{
// Attributes
public:
   Brickles* GetDocument();
// Implementation
public:
   virtual ~BricksView();
   virtual void OnUpdate(CView* pSender, LPARAM lHint = 0L,
     CObject* pHint = NULL);
   virtual void OnDraw(CDC* pDC);  // overridden to draw this view
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
protected:
   //{{AFX_MSG(BricksView)
   afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
   afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
   afx_msg void OnSize(UINT nType, int cx, int cy);
   afx_msg void OnMouseMove(UINT nFlags, CPoint point);
   afx_msg void OnTimer(UINT nIDEvent);
   //}}AFX_MSG
   DECLARE_MESSAGE_MAP()
};
```

Resulting Context:

An application architecture for handling Windows messages has been established by the interaction of the CCmdTarget subclasses. Unhandled messages are passed to a predetermined object.

Related Patterns:

This pattern is related to the **Map Windows messages to your application** pattern.

Pattern # 9: Associate Resources with Classes

Problem:

The Windows concept of a resource must be reconciled with the object-oriented design.

Context:

The architecture of the object-oriented application is built around classes and objects.

Forces:

There are many resources and this results in many classes.
Project tools and procedures are designed to work with classes.

Solution:

Use the AppWizard to create a resource entry and the ClassWizard to create a class that embodies that resource.

Resulting Context:

The appropriate resource entry is available for Windows and the class is available for the application.

Related Patterns:

This pattern is related to the **Provide low cost text labels** pattern and the **Give users control through menus** pattern.

Pattern # 10: Capture User Actions

Problem:

There is a need to capture small user decisions such as start, stop, exit actions or setting options on or off.

Context:

The resources for a user interaction have been encapsulated within an eay to use class.

Forces:

Many choices can be expressed as on/off choices.
Not much space available on the interface.

Solution:

Use instances of the **CButton** class to present options to users and to generate messages that the application will handle to react to the setting.

Resulting Context:

The application can present a number of small windows to the user. Each window has a label to indicate the option and can react to a mouse button press/release pair within its boundaries.

Related Patterns:

This pattern leads into the **Organize inputs into clusters** pattern.

Pattern # 11: Organize Inputs into Clusters

Problem:

There is a need to define windows that give a context to input requests made of the user.

Context:

When designing the user interface, only requests that are related should be presented at any one time.

Forces:

The user interface should remain uncluttered and easy to read.
Users want to be able to set as many features as possible in the application.
Computation can not proceed in the face of ambiguity.

Solution:

Use subclasses of **CDialog** to organize a set of instances of controls such as **CButton**s to present and record the user's response to information about a related set of options or a single option.

Resulting Context:

Input requests that are related can be grouped into a single dialog box, such as the printer setup box.

Related Patterns:

This is related to the **Capture user actions** pattern and leads into the **Give users control through menus** pattern.

Pattern # 12: Give Users Control through Menus

Problem:

There is a need to define menus that provide users with control of the application's actions.

Context:

In designing the user interface, menus are used to group commands in much the same way that dialog windows group inputs.

Forces:

Too many menus clutter the user interface.
Too few result in long lists of commands that are hard to remember.

Solution:

Create menu resources using the AppStudio and then create an instance of **CMenu** to load from the resource file.

Resulting Context:

Windows' menus are encapsulated within classes that associate the user's input with

Related Patterns:

This pattern is related to the **Create Main FrameWindow** pattern.

Pattern # 13: Provide Low Cost Text Labels

Problem:

There is a need to provide information that is not associated with input or output operations.

Context:

Users need to be given directions or informed of the status of the system.

Forces:

This information does not need to participate in the passing around of Windows mesages. The surrounding window should be as small as possible.

Solution:

Create instances of **CStatic** to provide labels or blank space between controls that collect input from the user.

Resulting Context:

The interface can now display arbitrary text in strategic locations.

Related Patterns:

This pattern is related to the **Associate resources with classes** pattern.

Pattern Relationships