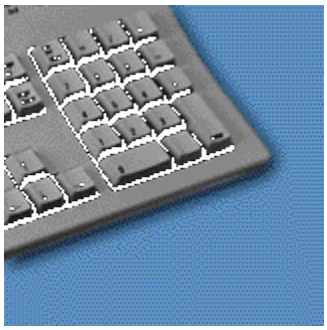


5. MFC의 디자인 패턴 (Design Patterns)

<장도비라>
디자인 패턴이란 문제를 해결하기 위해 상호 협조하는 클래스 간의 관계를 나타내는 용어이다. 그것은 클래스 간의 구조나 동작을 포함한다. 우리는 이 장에서 MFC의 골격을 이루는 중요한 디자인 패턴들을 살펴볼 것이다. MFC의 디자인 패턴을 이해하는 것은 MFC의 큰 그림을 이해하는데 도움이 된다. 이 장에서 우리는 MFC의 디자인 패턴을 통해 MFC의 큰 그림을 이해한 후, 6장부터 MFC의 세부를 이루는 부분들을 하나씩 살펴볼 것이다.
</장도비라>

이 장은 다음 장부터의 본격적인 MFC소스 코드 분석을 위해 MFC코드의 전반적인 설계 구조를 살펴보는 장이다. 이 장에서 아래의 사항들을 살펴볼 것이다.

- 디자인 패턴의 개념과 구체적인 패턴들
- MFC에 사용된 디자인 패턴들

디자인 패턴

<절도비라>
이 절에서는 디자인 패턴의 일반적인 개념을 살펴보고, 디자인 패턴에서 사용하는 용어의 뜻을 이해한다.
</절도비라>

다수의 온라인 사용자를 관리하기 위해 STL(standard template library)의 std::vector를 사용하기로 했다고 하자.

<저자 한마디>

이 책에서는 독자들이 STL에 자유하다고 가정한다.
</저자 한마디>

std::vector는 변형된 형태의 **리스트(list)**를 사용하며, 데이터를 정열(sort)하기 위해 **힙소팅(heap sorting)**을 사용한다고 가정하자. 이 때 리스트는 **자료구조(data structure)**를 일컫는 말이며, 힙소팅은 **알고리즘(algorithm)**을 일컫는 말이다.

이제 이러한 데이터를 DirectX 혹은 OpenGL을 기반으로 한 게임 프로그램의 클라이언트(client)에서 화면에 표시하는 작업을 생각해 보자. 이런! 독자들이 DirectX를 모르니까 그것은 불가능하다고? 그렇지 않다. 우리는 DirectX를 모르고도 많은 부분을 구현할 수 있는데, 그것은 바로 처리되는 내용과 내용을 그리는 부분을 별도의 클래스로 구현하는 것이다.

MFC에서는 이것을 다큐먼트/뷰 구조(document/view architecture)라 하는데, 3D 게임의 경우에는 '데이터와 렌더링의 분리'라고 할 수 있다.

다큐먼트/뷰 구조는 소프트웨어의 유지/보수와 디버깅을 획기적으로 개선하며, 디자이너와의 병행작업도 가능하게 한다.

다큐먼트와 뷰의 분리라는 개념은 자료구조도 알고리즘도 아니며 "**일반적 설계문제를 해결하기 위해 상호 교류하는 객체와 클래스에 대한 설명**"의 하나이다. 이렇게 객체 지향 언어의 클래스의 관계에 집중한 이러한 소프트웨어 구조를 **디자인 패턴(design pattern)**이라 한다□.

<여기서 잠깐>

Erich Gamma, et al., "Design Patterns", Addison Wesley, 1995
<여기서 잠깐>

디자인 패턴에 대한 용어 통일은 프로그래머들 간의 의사소통을 원활하게 하여 효율을 높이며, 생산성 향상에 많은 도움을 준다. 예를 들어 아래의 질문을 고려해 보자.

"클라이언트에서 사용자 리스트를 어떻게 구현하지?"

위 질문에 대해 서로가 **디자인 패턴**을 알고 있다면, 다음과 같이 대답할 수 있다.

"사용자 리스트와 DirectX 렌더링을 구조적으로는 브리지 패턴으로 설계해,

그리고, 렌더링 클래스를 관찰자 패턴(observer pattern)으로 구현해야 하겠군"

독자들은 이미 디자인 패턴을 알고 사용하고 있는데, 예를 들면 클래스의 **is-a 관계**와 **have-a관계**, 또한 **STL의 알고리즘(algorithm)**, **펑터(functor)**와 **아이터러터(iterator)** 같은 것들이다. 디자인 패턴에서는 STL의 펑터를 **전략 패턴(strategy pattern)**이라 한다.

is-a관계의 패턴과 펑터□의 패턴은 분류 기준이 다른데, 전자는 클래스의

<여기서 잠깐>

펑터(functor)란 함수 오브젝트(function object)를 말한다. 펑터는 매우 중요하고, 자주 사용되는 디자인 패턴인데, MFC의 소스 코드분석에는 사용되지 않으므로, 펑터에 대한 자세한 설명을 하지 않는다. 펑터는 클래스로 구현되며, 펑터가 지원하는 대상 클래스에서 호출해야할 규칙을 만족하는 멤버 함수들을 가진다. 펑터의 예로 std::vector<>의 두번째 파라미터로 전달되는 std::allocator<>가 있다. 사실 펑터는 C에서 함수 포인터로 구현한 단 하나의 콜백함수를 여러 콜백함수를 호출할 수 있도록 확장한 개념이다.

</여기서 잠깐>

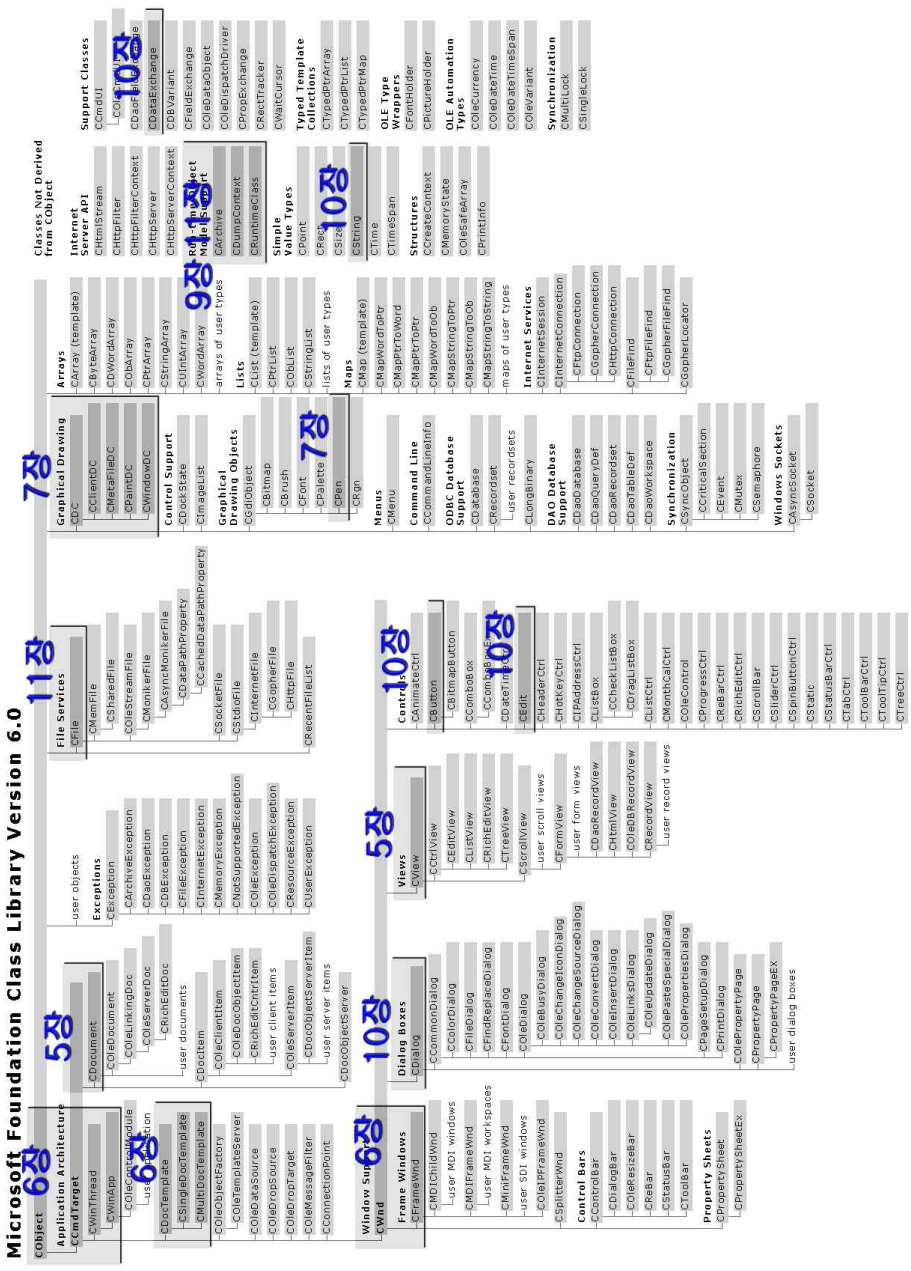
구조와 관계되어 있고, 후자는 행위(behavior)와 관련되어 있다. 각각을 **구조 패턴(structural pattern)**, **행위 패턴(behavioral pattern)**이라 한다.

4장에서 가상함수의 동작을 살펴보았다. 어떠한 클래스를 설계할 때 가상함수를 이용하여 전체 시스템의 일반적인 동작을 설계한 다음, 세부적인 구현과 기능의 확장을 연기할 수 있다. 이렇게 설계된 클래스 설계를 '**가상 패턴**'이라고 한다.

또한 상속으로 구현된 패턴을 '**상속 패턴**' 혹은 '**is-a 패턴**'이라고 하고, 포함으로 구현된 패턴을 '**포함 패턴**' 혹은 '**have-a 패턴**'이라 하자. 이러한 일반적이 패턴들 이외에, 우리는 이 장에서 MFC에서 매우 중요하게 사용된 몇개의 패턴들을 살펴볼 것인데, 그것들 중에는 구조를 기술하는 브리지 패턴(bridge pattern), 행위를 기술하는 스테이트 패턴(state pattern)과 관찰자 패턴(observer pattern)등이 있다. 또한 8장에서 HDC를 클래스로 유연하게 구현하기 위해 데코레이터 패턴(decorator pattern)을 사용할 것이다.

아래 그림에 이 장부터 분석을 시작할 MFC의 클래스를 표시하였다. [그림 5.1]에서 나타나듯이 MFC의 핵심 부분을 분석하고 구현할 것이다.

[아래의 그림은 수정해야 합니다]

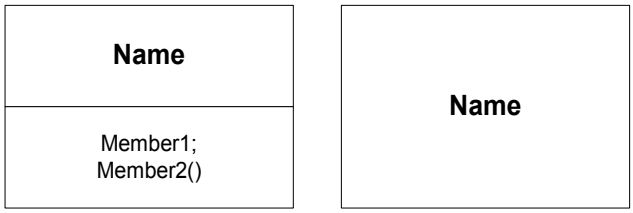


[그림 5.1] MFC의 클래스 다이어그램: 우리는 MFC의 핵심 부분을 직접 구현할 것이다.

클래스 다이어그램과 오토마타(automata)

<절도비라>
이 절에서는 디자인 패턴과 클래스 구조를 그리는 표준적인 방법과, 객체의 상태 변화를 표현하는 표준적인 방법인 오토마타에 대해서 살펴본다.
</절도비라>

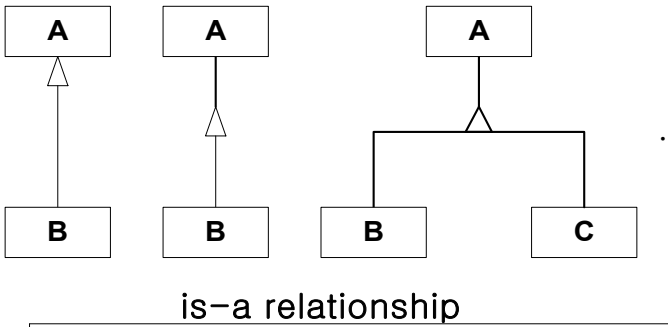
클래스 다이어그램(class diagram)을 그리기 위해서 사용하는 표기법을 통일하자. 필자는 많이 사용하는 표준을 사용하겠지만, 혼용되는 것들에 대해서도 허용할 것이다. 먼저 클래스는 아래 [그림 5.2]와 같이 표현한다.



[그림 5.2] 클래스의 표현

클래스는 사각형으로 표시하고, 위쪽에 클래스 이름을 아래쪽에는 멤버를 표시한다. 멤버를 표시할 필요가 없는 경우 클래스 이름만을 명시한다.

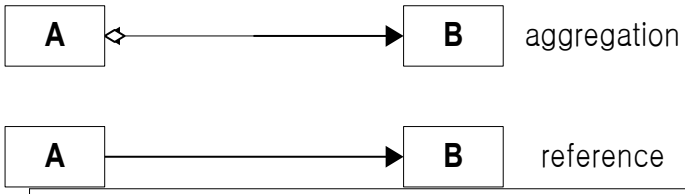
클래스의 **is-a 관계** - 상속관계 - 는 다음 [그림 5.3]과 같이 표현을 혼용한다.



[그림 5.3] 상속 관계의 표현

첫 번째 두 그림에서 B의 베이스 클래스는 A이며, 세 번째 그림은 A를 공통 조상으로 하는 상속(inheritance)을 표현한다.

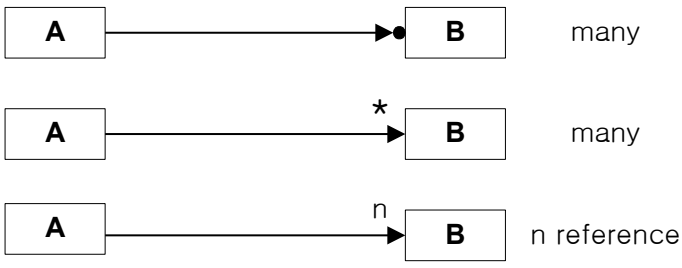
클래스의 have-a 관계 - 포함관계 - 는 다음 [그림 5.4]와 같이 표현한다.



[그림 5.4] 포함 관계의 표현: 포함(aggregation)은 B객체에 대한 생성과 소멸을 책임진다. B객체는 A객체가 만들어 질 때 같이 만들어질 수도 있고, 포인터로 선언된 경우, CreateBObject()등의 함수에 의해 A객체의 생성 후에 만들어질 수도 있다. 참조(reference)는 반드시 C++의 참조 기능으로 구현될 필요는 없다. 참조가 포인터로 구현된 경우, A객체는 B객체의 생성과 소멸을 책임지지 않는다. 이러한 경우 A 내부에서 B에 대한 참조 포인터는 m_pRefB; 처럼 멤버 함수이름에 명시적으로 표현되는 것이 바람직하며, std::auto_ptr<>처럼 스마트 포인터(smart pointer)로 구현할 수도 있다.

have-a 관계가 객체를 직접 포함(aggregation)하고 있는 경우는 시작하는 쪽에 다이아몬드를 표시한다. 포인터나 C++의 참조(reference)를 사용하는 경우는 시작하는 쪽에 다이아몬드를 표시하지 않는다. 그림은 A클래스가 B를 포함하고 있음을 의미한다.

대상이 한 개가 아닌 경우는 아래 [그림 5.5]처럼 끝나는 쪽에 원을 표시하거나, 개수를 직접 명시하는 표기법을 혼용한다.



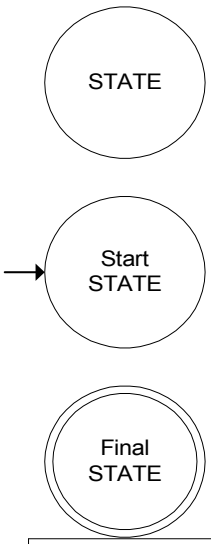
[그림 5.5] 일대 다 포함 관계의 표현

위 그림은 A 클래스가 B클래스 타입의 객체를 여러 개, 혹은 n개를 포함하는 것을 나타낸다.

오토마타

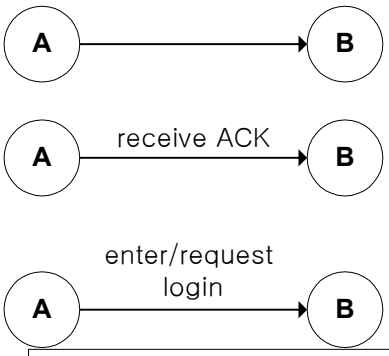
클래스 다이어그램은 클래스 간의 구조 관계를 표현하지만, 각 클래스간의 행위 관계를 표현하지는 못한다. 행위 관계를 표현하는 전통적인 방법으로 **상태 변화도(state transition diagram)**, 즉 **오토마타(automata)**를 사용할 수 있다. 오토마타는 입력에 반응하는 출력을 결정하기 위해, 상태, 상태천이, 입력과 출력을 그림으로 표현한다.

오토마타에서 상태(state)는 아래 [그림 5.6]처럼 원안에 표시한다.



[그림 5.6] 상태의 표현

시작 상태(start state)는 나오는 곳이 없는 화살표를 추가하여 표시한다. 종료 상태(final state)는 이중 원(double circle)로 표시한다. 상태 변화(state transition)는 상태와 상태를 연결하는 화살표로 표시한다. 아래 [그림 5.7]을 보자.



[그림 5.7] 상태 변화의 표현

위 그림은 모두 상태 A에서 상태 B로의 상태 변화를 표시한다. 상태 변화를 일으키는 입력을 화살표 위에 표시하고, 출력을 입력 다음에 /(slash) 이후에 표시한다.

두 번째 그림은 A상태에서 서버(server)로부터 ACK를 받은 경우, B상태로 변화함을 나타낸다. 세 번째 그림은 A상태에서 enter키를 입력 받은 경우, 서버에 로그인을 요구하고 B상태로 변화함을 나타낸다.

브리지 패턴(bridge pattern)

<절도비라>

이 절에서는 자세한 구현과 추상화라는 개념을 분리해서 표현하는 브리지 패턴에 대해서 살펴본다. 그리고 브리지 패턴의 예로 다큐먼트와 뷰를 구현해 보고, 13장에서 구현할 직렬화가 브리지 패턴임을 이해한다.

</절도비라>

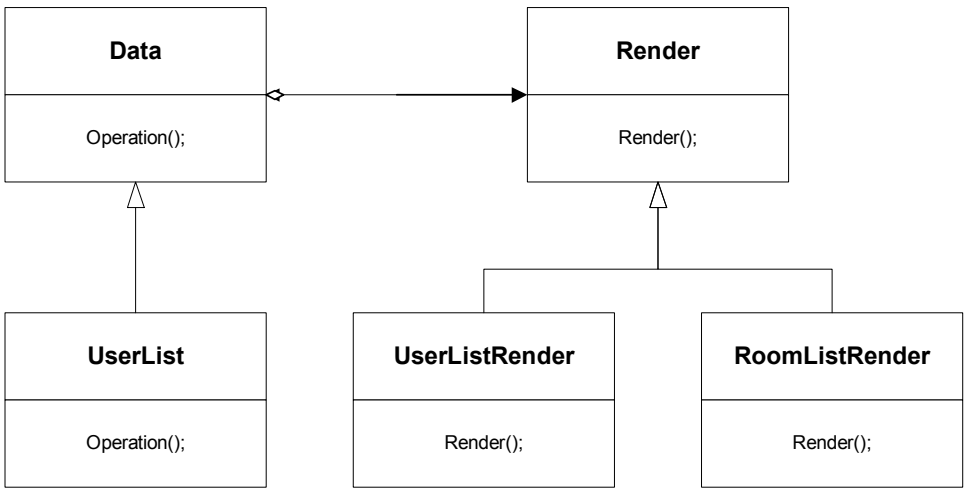
브리지 패턴은 구현(implementation)과 추상화 개념(abstraction)을 분리하는 구조 패턴이다(다르게는 핸들/바디(handle/body), 중복된 일반화(nested generalization)라고도 한다).

예를 들면 온라인 게임의 클라이언트에서 사용자 리스트를 화면에 출력하는 기능을 생각해 보자. 게임이 DirectX를 사용한다면 TL 버텍스(transformed and lighted vertex)를 사용해서 한 사용자 아이템을 렌더링 할 수 있다. 이제 사용자 리스트의 관리와 화면 출력을 별도의 클래스로 구현한다. 이러한 구현

은 다음과 같은 이점이 있다.

1. 렌더링과 별도로 사용자 리스트를 구현함으로써 충분한 테스트와 디버깅을 쉽게 해 볼 수 있다.
2. 인터페이스와 구현의 결합도가 약해진다. 이것은 구현 방식에 얽매이지 않고 인터페이스를 설계하거나, 인터페이스에 얽매이지 않고 구현을 설계할 수 있다.
3. 확장이 쉽다. 데이터와 렌더링을 독립적으로 확장할 수 있다.
4. 구현을 은닉할 수 있다. 인터페이스 설계자는 데이터의 내부 구조의 변화를 예상하지 않아도 된다.

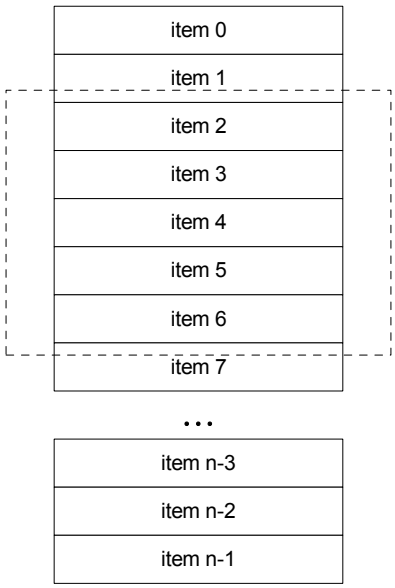
예의 경우 클래스 구조는 [그림 5.8]과 같은 브리지 패턴으로 표현한다.



[그림 5.8] 데이터와 렌더링의 브리지 패턴: 데이터와 표현 방식을 분리해서 구현함으로써 효율을 높인다.

UserList 클래스와 UserListRender 클래스는 별도로 구현된다. 이제 각각은 상호 의존적이지 않게 유지/보수가 가능하다.

이제 UserList쪽을 구현해 보자. 가변적인 사용자 리스트를 화면에 표시할 때 스크롤을 고려한다. 아래 [그림 5.9]를 보자.

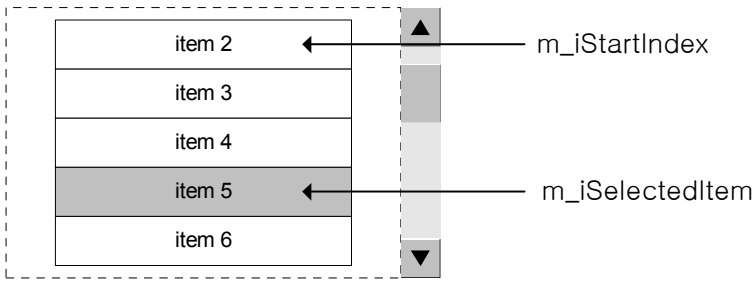


[그림 5.9] 사용자 리스트의 관리: std::vector를 이용하여 사용자 리스트를 관리한다. 현재 항목의 개수는 n개라 하자. 점선으로 표시된 부분이 화면에 표시될 부분이다.

그림에서 점선으로 표시될 부분이 화면에 표시될 부분이라면, 다음과 같은 변수를 유지해야 한다. 그것은 다음과 같다.

```
현재 선택된 아이템(m_iSelectedItem)
항목 표시의 시작 위치(m_iStartIndex)
표시될 항목의 수(m_numItemToDisplay)
```

아래 [그림 5.10]을 보자.



[그림 5.10] 사용자 리스트의 화면 표시: 화면 표시를 위해 최소한 3개의 변수를 유

지한다. 마우스와 키보드의 사용자 인터페이스를 위한 작업은 브리지 패턴의 다른 곳에서 구현될 것이다.

클래스는 [예제 5.1]과 같은 구조가 될 것이다.

[예제 5.1] class KListBox

```
template<class T>
class KListBox
{
public:
    typedef std::vector<T>          VTYPE; ///< item container type
    typedef VTYPE::iterator         VITOR; ///< item iterator

    VTYPE       m_vector;           ///< vector container
    int         m_iSelectedItem;    ///< selected item index
    int         m_iStartIndex;      ///< display start index
    int         m_numItemToDisplay; ///< number of items to display
};
```

KListBox는 또한 std::vector가 제공하는 아이터러터를 래퍼 클래스(wrapper class) - 혹은 데코레이터 패턴으로 제공할 것이다. 구현된 KListBox를 [예제 5.2]에 리스트하였다.

[예제 5.2] 05_KListBox.h

```
////////////////////////////////////////////////
//
/// @file   KListBox.h
///         interface for the KListBox class of 'vector' project.
/// @author seojt@kogsoft.com
/// @since  2003-07-05 am 08:43:31
///

#if !defined(_KListBox_Defined_)
#define _KListBox_Defined_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
```

```
#include <vector>

////////////////////////////////////////////////.
/// @class  KListBox
///
/// @see
///
template<class T>
class KListBox
{
public:
    typedef std::vector<T>          VTYPE; ///< item container type
    typedef VTYPE::iterator         VITOR; ///< item iterator

    VTYPE       m_vector;           ///< vector container
    int         m_iSelectedItem;    ///< selected item index
    int         m_iStartIndex;      ///< display start index
    int         m_numItemToDisplay; ///< number of items to display

public:
    /// constructor.
        KListBox()
    {
        m_iSelectedItem     = 0;
        m_iStartIndex       = 0;
        m_numItemToDisplay  = 9;
    }//KListBox()

    /// destructor.
        virtual ~KListBox()
    {
        Clear();
    }//~KListBox()

    /// get begin item.
    /// @return VITOR : iterator
    VITOR Begin()
    {
        return m_vector.begin();
    }//Begin()

    /// get last item.
    /// @return VITOR : iterator
```

```
    VITOR End()
    {
        return m_vector.end();
    }//End()

    /// clear all items in the vector container.
    ///  @note     PreClear() and PostClear() will be called before
container clearing
    void Clear()
    {
        PreClear();
        m_vector.clear();
        PostClear();
    }//Clear()

    /// Prepare m_vector.clear().
    virtual PreClear(){}

    /// post processing after calling m_vector.clear().
    virtual PostClear(){}

    /// insert item to the end of container.
    /// @param  item : item to insert
    void Insert(T item)
    {
        m_vector.push_back( item );
    }//Insert()

    /// insert item at specified position.
    /// @param  itor : insert before position
    /// @param  item : item to insert
    void Insert(VITOR itor, T item)
    {
        m_vector.insert( itor, item );
    }//Insert()

    /// access m_vector[index].
    /// @param  index : index to container
    T& At(int index)
    {
        return m_vector[index];
    }//At()
```

```
    /// access m_vector[index].
    /// @param  index : index to container
    T& operator[](int index)
    {
        return m_vector[index];
    }//operator[]()

    /// find item by value.
    /// @param  item : item to find
    /// @return VITOR : found iterator\n
    ///         - NULL when 'item' isn't found
    VITOR FindByValue(T item)
    {
        for (VITOR itor = m_vector.begin(); itor != m_vector.end();
++itor)
        {
            if ( item == (*itor) ) return itor;
        }//for
        return NULL;
    }//FindByValue()

    /// erase item by value.
    /// @param  item : item to erase
    /// @return bool : 'true' when erased
    bool EraseByValue(T item)
    {
        VITOR itor = FindByValue( item );
        if ( itor ) m_vector.erase( itor );

        return itor != NULL;
    }//Erase()

    bool EraseByIndex(int iItem)
    {
        int index = 0;
        for (VITOR itor = m_vector.begin(); itor != m_vector.end();
++itor)
        {
            if ( index == iItem )
            {
                Erase( itor );
                return true;
            }//if
```

```
        }//for

        return false;
    }//EraseByIndex()

    /// direct erase.
    /// @param  itor : iterator to erase
    void Erase(VITOR itor)
    {
        m_vector.erase( itor );
    }//Erase()

    /// erase ranged items.
    /// @param  itorBegin : begin iterator
    /// @param  itorEnd : end iterator
    void Erase(VITOR itorBegin, VITOR itorEnd)
    {
        m_vector.erase( itorBegin, itorEnd );
    }//Erase()

    /// check whether container is empty.
    /// @return bool : 'true' if containter is empty
    bool IsEmpty()
    {
        return m_vector.empty();
    }//IsEmpty()

    /// get number of items in the container.
    /// @return int : number of items in the m_vector[]
    int GetSize()
    {
        return m_vector.size();
    }//GetSize()

    /// reserve items.
    /// @param  size : number of items to reserve
    void Reserve(int size)
    {
        m_vector.reserve( size );
    }//Reserve()

public:
    /// set selected item index.
```

```
    /// @param  iItem : highlighted item index
    void SetSelected(int iItem)
    {
        m_iSelectedItem = iItem;
    }//SetSelected()

    int GetSelected()
    {
        return m_iSelectedItem;
    }//GetSelected()

    /// check whether iSelected is highlighted item.
    /// @param  iSelected : test index
    /// @return bool : true if 'iSelected' is highlighted
    bool IsSelected(int iSelected)
    {
        return m_iSelectedItem == iSelected;
    }//IsSelected()

    /// set display start index.
    /// @param  iIndex : item index started to display
    int SetDisplayStart(int iIndex)
    {
        m_iStartIndex = iIndex;
        return m_iStartIndex;
    }//SetDisplayStart()

    /// get display start index.
    /// @return int : start index of displaying
    int GetDisplayStart()
    {
        return m_iStartIndex;
    }//GetDisplayStart()

    /// set number of items to display.
    /// @param  iSize : number of items to display
    void SetNumDisplay(int iSize)
    {
        m_numItemToDisplay = iSize+1;
    }//SetNumDisplay()

    int GetNumDisplay()
    {
```

```
        return m_numItemToDisplay-1;
    }//GetNumDisplay()

    VITOR BeginDisplay()
    {
        if ( m_iStartIndex >= GetSize() ) return m_vector.end();
        return &m_vector[m_iStartIndex];
    }//BeginDisplay()

    VITOR EndDisplay()
    {
        int endIndex = m_iStartIndex + m_numItemToDisplay - 1;
        if ( endIndex >= GetSize() ) return m_vector.end();
        return &m_vector[endIndex];
    }//EndDisplay()
};//class KListBox

#endif // !defined(_KListBox_Defined_)
```

이중 포인터 등의 처리를 위해 데이터를 지우기 전과 후에 호출되는 가상함수 PreClear()와 PostClear()를 제공한다. 데이터를 접근하기 위해 At() 혹은 operator[]()를 사용할 수 있다. std::vector를 알고 있는 독자들은 쉽게 소스를 따라 갈 수 있을 것이다. 소스가 이해되지 않는 독자들은 부록 CD-ROM에 첨부된 STL문서를 읽어 보기 바란다.

이제 브리지 패턴의 데이터 쪽을 완성했다. 이 클래스가 동작하는지 테스트 루틴을 작성해 보자. 아래 [예제 5.3]를 보자.

[예제 5.3] 05_KListBox.h의 테스트 루틴

```
#include <iostream>
#include "05_KListBox.h"

using std::cout;
using std::endl;

void main()
{
        // Create a vector object of integers
        KListBox<int>   kListBox;
```

```
        // Fill the vector with 3 different elements
    for (int i=0; i<20; ++i)
    {
            kListBox.Insert(i);
    }//for

    int iStart = kListBox.SetDisplayStart( 15 );
    kListBox.SetSelected( 17 );
        // Now loop and print out all the element values
    for(KListBox<int>::VITOR itor = kListBox.BeginDisplay(); itor !=
kListBox.EndDisplay(); ++itor)
    {
        if ( kListBox.IsSelected(iStart) )
            cout << "* ";
        else
            cout << "  ";
                cout << "element value = " << (*itor) << endl;
        ++iStart;
    }//for
}//main()
```

출력 결과는 다음과 같다.

```
  element value = 15
  element value = 16
* element value = 17
  element value = 18
  element value = 19
```

MFC는 구조적으로 다큐먼트(데이터)와 뷰(렌더링)의 약한□ 브리지 패턴을 사용한다.

<여기서 잠깐>

브리지 패턴은 추상화와 구현을 분리한 구조 패턴이다. 필자는 다큐먼트에 구현을 뷰에 추상화를 대응했다. 하지만, 다큐먼트/뷰 구조는 특정한 구조 패턴으로 분류할 수 없는 것처럼 느껴진다. MFC의 디자인 패턴 중 확실하게 브리지 패턴을 사용하는 직렬화(Serialization)에 관해서는 13장에서 살펴볼 것이다.

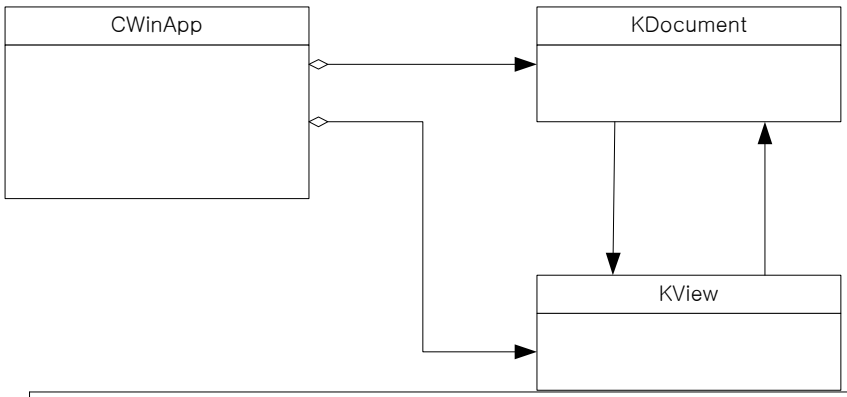
</여기서 잠깐>

데이터를 나타내는 다큐먼트 클래스와 데이터의 표시를 담당하는 뷰 클래스는 별도의 클래스로 구현되어 있고, 하나의 다큐먼트에 다수의 뷰를 연결하는 것이 가능하다.

다큐먼트와 뷰의 상호 작용과 이에 관계된 멤버 함수를 이해하는 것은 MFC 프로그래밍의 넘어야 할 산이다. MFC는 브리지 패턴이외에도 많은 디자인 패턴을 사용한다. MFC는 엄청나게 큰 잘 설계된 디자인 패턴의 덩어리이다. 처음 MFC의 진입을 어렵게 만드는 것은 이러한 디자인 패턴에 대한 이해의 부족에서 출발한다. MFC의 디자인 패턴을 이해하는 것은 윈도우 프로그래밍의 넘어야 할 산이다.

다큐먼트와 뷰의 구현

이제 다큐먼트와 뷰를 구현해 보자. 필자는 화면 출력을 위해 GDI가 아닌 콘솔(console)을 사용할 것이다.



[그림 5.10b] 다큐먼트와 뷰: CWinApp는 다큐먼트와 뷰를 생성한다. 다큐먼트와 뷰는 서로를 참조하며, 다큐먼트는 데이터 관리를, 뷰는 다큐먼트가 관리하는 데이터의 화면 출력을 담당한다.

설계할 다큐먼트 클래스는 KDocument이다. 다큐먼트 클래스는 오직 데이터 관리에만 집중한다. 아래의 소스를 보자.

```
#include "KListBox.h"
#include "KView.h"

class KDocument
{
```

```
public:
    typedef KListBox<int>::VITOR    KDITOR;

private:
    KListBox<int>   m_kListBox; // (1)
    KView*          m_pView; //(2)

public:
    KDocument();
    virtual ~KDocument();

    void Insert(int iData);
    void SetView(KView* pView);
    KView* GetView(); // (3)
    KListBox<int>* GetListBox();
};//class KDocument
```

다큐먼트 클래스는 리스트 박스를 유지한다(1). 다큐먼트에 연결된 뷰에 관한 정보를 얻기 위해 뷰 객체에 대한 포인터를 멤버로 가지며(2), 뷰를 얻는 멤버 함수가 있어야 한다(3).

이제 뷰를 구현한다. 뷰는 데이터 관리에는 상관하지 않고 다큐먼트가 관리하는 데이터를 표현하기 위한 멤버 만을 가진다. 설계할 뷰 클래스의 이름은 KView이다. 아래 소스를 보자.

```
class KDocument;
class KView
{
private:
    KDocument*  m_pDocument; // (1)
    int         m_iStart; // (2)
    int         m_iDisplayStart;
    int         m_iSelected;

public:
    KView(KDocument* pDoc);
    virtual ~KView();

    void SetDocument(KDocument* pDoc);
    KDocument* GetDocument(); // (1)
    void SetDisplayStart(int iStart);
    void SetSelected(int iSelected);
```

```
    virtual void Draw();
};//class KView
```

뷰는 연결된 다큐먼트를 위한 멤버를 가진다(1). 다큐먼트 객체에 대한 포인터를 제외한 다른 뷰의 모든 멤버는 '보여주기'를 위해서 필요한 것들이다(2).

이제 다큐먼트와 뷰를 포함하는 응용 프로그램 클래스를 만들고 이를 테스트해 볼 수 있다. 응용 프로그램 클래스를 CWinApp라 하자. [예제 5.4]에 소스를 리스트하였다.

[예제 5.4] class CWinApp

```
#include <iostream>
#include <assert.h>
#include "KDocument.h"
#include "KView.h"
//#include <conio.h>

class CWinApp
{
private:
    KDocument*  m_pDoc;
    KView*      m_pView;

public:
    CWinApp()
    {
        m_pDoc  = new KDocument(); // (1)
        m_pView = new KView( m_pDoc ); // (2)
        m_pDoc->SetView( m_pView ); // (3)
    }//CWinApp()

    CWinApp(const CWinApp& right)
    {
        assert( !"fail" );
    }//CWinApp()

    ~CWinApp()
    {
        delete m_pDoc;
        delete m_pView;
```

```
    }//CWinApp()

    void InitInstance()
    {
        for (int i=0; i<20; ++i)
        {
            m_pDoc->Insert( i ); // (4)
        }//for
    }//InitInstance()

    void OnDraw()
    {
        m_pView->SetDisplayStart( 15 ); // (4)
        m_pView->SetSelected( 17 );
            m_pView->Draw();
    }//OnDraw()
};//class CWinApp

CWinApp theApp;

void main()
{
    theApp.InitInstance();
    theApp.OnDraw();
    //getch();
}//main()
```

응용 프로그램 객체의 생성자에서 다큐먼트와(1) 뷰 객체를 만들고 뷰 객체에 다큐먼트를 설정한 다음(2), 다큐먼트 객체에 뷰를 설정한다(3). 데이터를 조작하는 것과 데이터를 표현하는 것이 분리되었음(4)에 주목하라. 이제 다큐먼트/뷰 구조를 가지고 뷰의 변경없이 다큐먼트를 변경하거나, 다큐먼트의 변경없이 뷰를 변경하는 것이 가능하다!

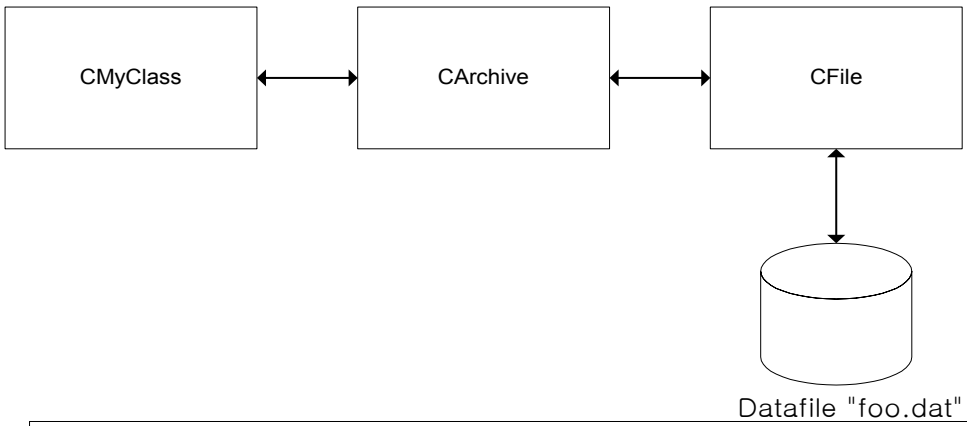
직렬화(serialization)

MFC의 **직렬화(serialization)**는 브리지 행동 패턴으로 구현된다. 직렬화란 객체의 내용을 디스크 파일이나 네트웍 소켓으로 입/출력하는 것□을 말한

<여기서 잠깐>

직렬화란 출판하다(serialize)란 의미를 가지고도 있지만, "차례대로"란 의미로 이해하는 것이 바람직하다. 직렬화를 하는 대부분의 이유가 파일 입/출력이나 소켓 입/출력을 위해서이기 때문인 것은 맞지만, 직렬화의 대상이 무엇이든 여러 객체들을 **"차례대로(serial)"** 만드는 것이 직렬화이다. 대상은 파일, 메모리, 네트웍 스트림(network stream) 등 아무것이나 될 수 있다.
</여기서 잠깐>

다. MFC는 다큐먼트 객체의 내용을 저장하기 위해, 파일을 사용하지 않고 아카이브(archive)라는 다리(bridge)를 사용한다. 직렬화는 파일로 구현되지만, 사용자에게 아카이브로 추상화된다.



[그림 5.11] 브리지 패턴으로 구현한 직렬화: 파일로 구현한 객체 입출력은 아카이브로 추상화되어 제공된다. 사용자는 객체를 저장하기 위해 아카이브라는 다리(bridge)만을 사용한다. 파일이라는 구현을 아카이브가 추상화한다는 개념에서 직렬화는 브리지 패턴이지만, 파일 접근을 통제하기 위해 파일 입출력을 대리하는 대리자로써 아카이브를 본 다면, 대리자 패턴(proxy pattern)이라고 할 수도 있다.

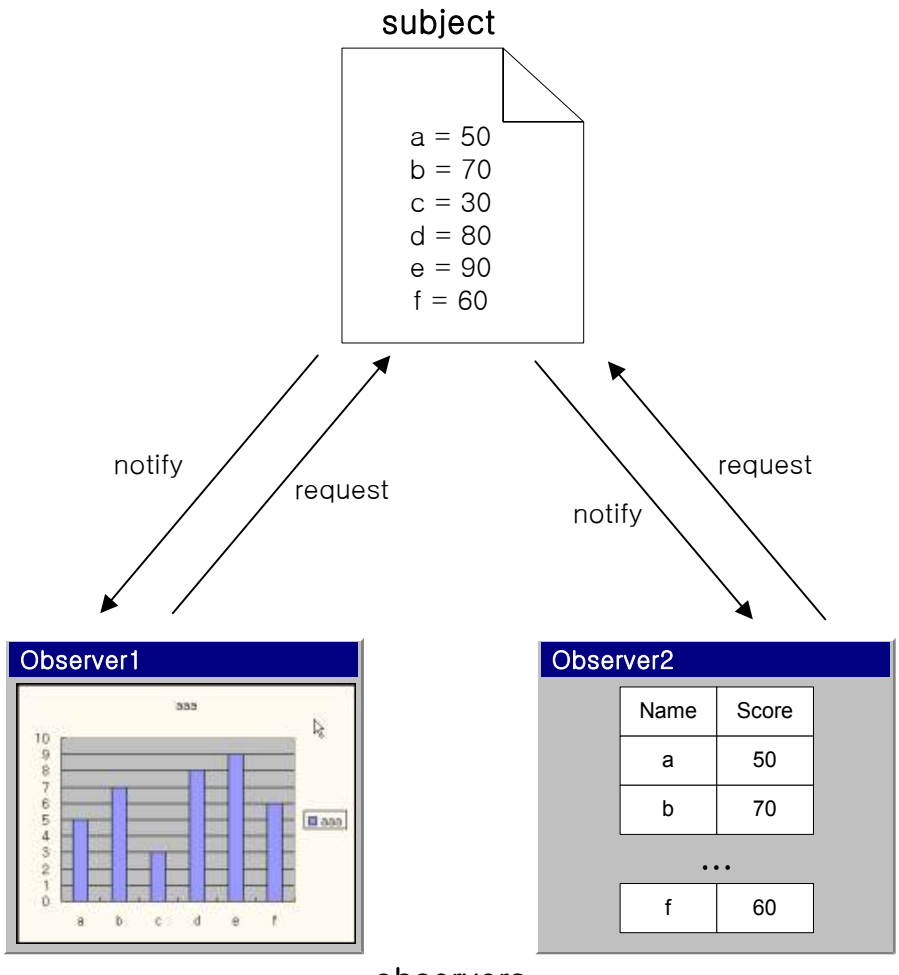
직렬화의 자세한 구현은 13장에서 살펴 볼 것이다.

다큐먼트/뷰 구조가 구조 패턴으로는 브리지 패턴이라면 행위 패턴으로는 어떤 패턴에 해당할까? 다큐먼트/뷰 구조는 대표적인 관찰자 패턴(observer pattern)이다.

관찰자 패턴(observer pattern)

<절도비라>
이 절에서는 관찰자 패턴의 일반적인 동작과 구조를 살펴보고, 연관된 함수들을 이해한다. 다큐먼트와 뷰가 대표적인 관찰자 패턴임을 이해한다.
</절도비라>

관찰자 패턴은 하나의 주제(subject)와 연관된 여러 관찰자(observer)의 동작을 표현하는 행위 패턴이다. 아래 [그림 5.12]를 보자.



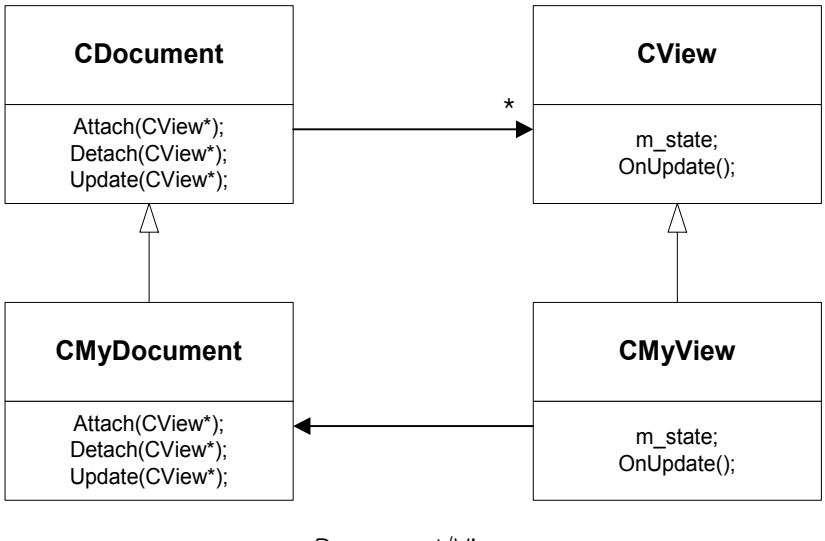
[그림 5.12] 관찰자 패턴: 한 주제(subject)를 관찰하는 여러 관찰자는 통지를 처리하는 통지 핸들러(notification handler)를 가진다. 또한, 주제를 수정하여 다른 관찰자에게 통지를 요청하기도 한다.

[그림 5.12]는 한 주제를 관찰하는 두 관찰자가 관찰 결과를 서로 다른 형식

으로 표현한다. 주제는 내용이 바뀔 때 각각의 관찰자에게 주제가 바뀌었음을 통지(notify)하고, 한 관찰자는 내용을 변경하고 다른 관찰자로의 통지를 요청(request)할 수 있다.

이와 같은 구조는 주제 클래스와 관찰자 클래스 각각을 재사용 할 수 있으며, 한 객체에 가해진 변경으로 다른 객체를 변경해야하는 작업을 자동화한다.

MFC의 다큐먼트/뷰 구조는 대표적인 관찰자 패턴의 예이다. 주제 클래스에 해당하는 부분이 다큐먼트 클래스이고, 관찰자 클래스에 해당하는 부분이 뷰 클래스이다. 다큐먼트/뷰 구조는 다음 [그림 5.13]과 같은 클래스 다이어그램으로 표현할 수 있다.



[그림 5.13] 다큐먼트/뷰 구조의 클래스 다이어그램: 하나의 다큐먼트는 여러 개의 뷰에 연결된다. 다큐먼트의 Update()명령에 대해, 뷰에서는 OnUpdate() 핸들러를 가진다.

다큐먼트 클래스는 다수의 뷰 클래스를 관리한다. 다큐먼트는 뷰 객체를 추가하거나 삭제하는 멤버 함수 Attach(), Detach()를 가지고, 뷰에게 업데이트를 요청하는 Update(), UpdateAllViews() 멤버 함수를 가진다. 뷰는 업데이트의 요구에 대응하는 이벤트 핸들러 OnUpdate()를 가진다. 응용 프로그램의 구현은 다큐먼트와 뷰의 베이스 클래스를 상속받아 구현하며, 뷰 클래스는 자신에게 연결된 다큐먼트를 참조할 수 있어야 한다.

스테이트 패턴(state pattern)

<절도비라>

이 절에서는 문제를 해결하는 과정을 상태로 구분하여 행동을 표현하는 스테이트 패턴에 대해서 살펴본다. 또한, 함수 포인터 테이블을 사용해서 이를 구현하는 방법을 살펴본다. 스테이트 패턴은 6장의 핵심이 될 내용이므로 꼼꼼하세 살펴보고 완벽하게 이해하기 바란다.

</절도비라>

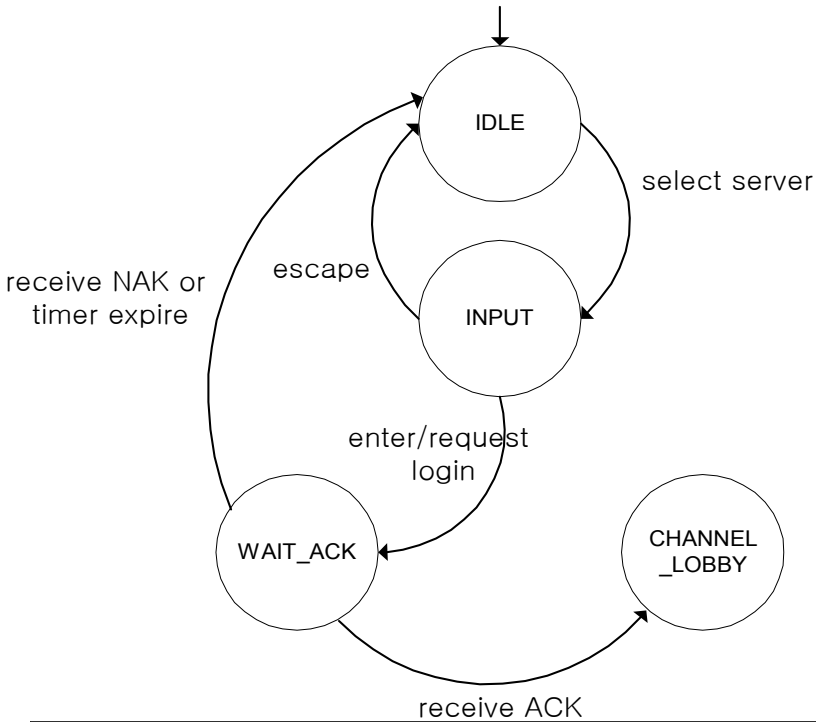
이제 행위 패턴의 하나인 **스테이트 패턴(state pattern)**을 살펴 볼 것이다. 스테이트 패턴을 멤버 함수의 포인터 테이블로 구현하는 것은 널리 알려진 구현 방식인데, MFC는 윈도우 메시지를 처리하기 위해 이 방법을 사용하므로 잘 이해해 두기 바란다.

객체의 내부 상태에 따라 객체의 행동을 변경해야 하는 경우가 있다. 사실 모든 객체는 내부 상태를 가진다고 할 수 있는데, 구분되는 상태가 없는 객체조차도 한 개의 상태를 가지기 때문이다.

3D 온라인 게임의 클라이언트가 서버에 접속하는 사용자 인터페이스(UI: user interface)를 생각해 보자. 사용자는 화면에 표시된 서버에서 한 서버를 선택한다. 서버에 접속하기 위한 계정을 묻는 대화상자가 뜨고, 사용자가 계정을 입력하면 서버에 접속을 시도한다. 얼마후 서버에서 접속 허가가 떨어지면, 사용자는 서버에 접속하여 채널을 선택하는 화면으로 이동한다.

자 이제 서버 리스트에서 서버를 선택하고, 대화상자가 출력되고, 채널 선택 화면으로 바뀌기 전까지의 UI를 한 클래스가 담당한다고 가정하자. 이 클래스는 각 상태마다 화면에 적절한 렌더링을 수행해야 한다.

독자들은 Render() 함수에서 if문을 이용하여 각 상태에 따라 적절한 렌더링 함수의 호출을 생각했을 것이다. 물론 그것이 해결책이지만, 스테이트 패턴을 이용하면 객체 내부가 좀더 구조화되고 유지/보수가 쉬운 코드를 작성할 수 있다. 아래 [그림 5.14]를 보자.



[그림 5.14] 서버 로그인: 계정(account)을 입력하면 서버로부터 응답을 기다리는 상태가 된다. 서버로부터 응답을 받으면 채널 선택상태로 바뀐다.

설계할 CAccount 클래스는 구분된 세 개의 상태를 가진다.

IDLE: 사용자는 서버 리스트에서 한 서버를 선택할 수 있다.
INPUT: 사용자가 서버를 선택한 경우, 서버 계정을 묻는 대화상자가 출력된다. 사용자는 사용자 ID와 비밀번호를 입력한다. 사용자가 확인 버튼을 선택하면 서버에 로그인을 요청하는 패킷(packet)을 전송한다.
WAIT_ACK: 서버에서 응답이 올 때까지 기다린다. 서버가 승낙(ACK: acknowledge) 패킷을 전송한 경우, 채널을 선택하는 다른 UI 화면으로 이동한다.

게임 루프는 계속해서 CAccount::Render()를 호출한다. 예를 들면 서버로부터 응답을 기다리는 동안에도 계속해서 CAccount::Render()는 호출되므로, 적절한 렌더링을 수행해야 한다(서버로부터 응답을 기다리는 동안 응용 프로그램이 멈추도록 하는 것은 좋지 않은 방법이다).

각 상태에서 렌더링을 수행하는 함수의 이름이 다음과 같다고 하자.

```
    void OnIdle();
    void OnInput();
    void OnWaitAck();
```

이제 Render()함수의 내부에 이 함수들의 테이블을 만든다.

```
void CAccount::Render()
{
    // state map table
    static void (CAccount::*mfp[])() =
    {
        CAccount::OnIdle,
        CAccount::OnInput,
        CAccount::OnWaitAck
    };//CAccount::map[]
}
```

이 테이블은 모든 객체마다 동일하므로 static선언을 한다. 상태를 유지하는 멤버 변수를 m_iState라하면, m_iState를 이용하여 각 상태의 렌더링 함수를 호출할 수 있다.

```
void CAccount::Render()
{
    // state map table
    static void (CAccount::*mfp[])() =
    {
        CAccount::OnIdle,
        CAccount::OnInput,
        CAccount::OnWaitAck
    };//CAccount::map[]

    if ( m_iState >= 0 && m_iState < STATE_MAX )
    {
        (this->*mfp[m_iState])();
    }//if
}
```

이러한 구현은 상태의 추가 제거가 쉽고, 각 상태의 렌더링 함수가 구조화되며, 상태의 수에 상관없이 즉시 호출되므로 빠르다.

아래에 [예제 5.5]를 리스트하였다. 사용자는 '1', '2', '3' 혹은 Esc를 누를 수 있다. 각 숫자키는 account 객체의 상태를 변경한다. 프로그램은 0.5초 단위로 각 상태에 해당하는 렌더링을 수행한다. 사용자가 Esc를 입력하면 프로그램은 종료한다.

[예제 5.5] 스페이트 패턴의 구현

```
#include <iostream>
#include <conio.h>
#include <windows.h>

#define KEY_ESC     27

using std::cout;
using std::endl;

class CAccount
{
public:
    enum
    {
        STATE_IDLE,
        STATE_INPUT,
        STATE_WAIT_ACK,

        STATE_MAX
    };

public:
    int     m_iState;

    CAccount();
    void OnIdle();
    void OnInput();
    void OnWaitAck();

    void SetState(int iState);
    void Render();
};//class CAccount

CAccount::CAccount()
```

```
{
    m_iState = STATE_IDLE;
}

void CAccount::OnIdle()
{
    cout << "OnIdle()" << endl;
}

void CAccount::OnInput()
{
    cout << "OnInput()" << endl;
}

void CAccount::OnWaitAck()
{
    cout << "OnWaitAck()" << endl;
}


void CAccount::SetState(int iState)
{
    m_iState = iState;
}

void CAccount::Render()
{
    // state map table
    static void (CAccount::*mfp[])() =
    {
        CAccount::OnIdle,
        CAccount::OnInput,
        CAccount::OnWaitAck
    };//CAccount::map[]

    if ( m_iState >= 0 && m_iState < STATE_MAX )
    {
        (this->*mfp[m_iState])();
    }//if
}

void main()
{
```

```
    CAccount    account;
    int         ch = 0;

    while ( ch != KEY_ESC )
    {
        if ( kbhit() )
        {
            ch = getch();
            if ( ch == '1' )
                account.SetState( CAccount::STATE_IDLE );
            else if ( ch == '2' )
                account.SetState( CAccount::STATE_INPUT );
            else if ( ch == '3' )
                account.SetState( CAccount::STATE_WAIT_ACK );
        }//if

        Sleep( 500 );
        account.Render();
    }//while
}//main()
```

스테이트 패턴을 구현하기 위해 사용된 기교를 **이벤트 패턴(event pattern)**에 사용할 수 있으며, 이것은 매우 중요하다. 아래와 같은 예를 생

<저자 한마디>

Erich의 책 "Design Patterns"에서 이벤트 패턴이라는 용어는 사용하지 않았지만, MFC의 메시지 맵을 표현하는 가장 적절한 패턴 용어라 생각한다. 가장 유사한 행위 패턴을 **책임의 체인 패턴(chain of responsibility)**이라고 할 수는 있지만, 그것과는 다르다. 필자는 이하 설명을 이벤트 패턴이라고 한다.

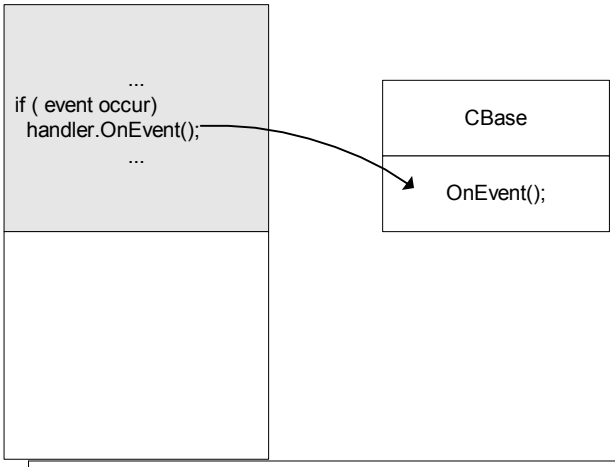
</저자 한마디>

각해 보자.

어떤 라이브러리를 사용하는 경우, 라이브러리에 이미 존재하는 코드에서 호출하는 이벤트 핸들러(event handler)를 생각해 볼 수 있다. 대응하는 이벤트 핸들러의 코드가 CBase::OnEvent()에 이미 작성되어져, 이 코드 역시 라이브러리의 일부로 제공된다.

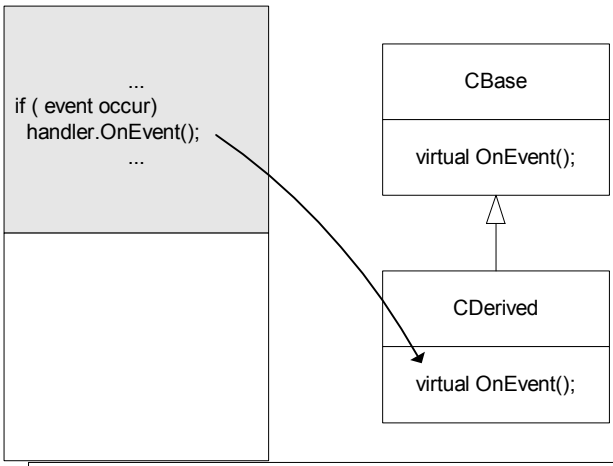
이제 이벤트 핸들러에 추가적인 작업을 해야 한다고 하면, 어떻게 이 문제

를 해결해야 하는가?



[그림 5.15] 이미 정의된 코드의 이벤트 핸들러 호출: 라이브러리 코드의 일부가 이벤트 핸들러를 호출한다. 이벤트 핸들러 역시 라이브러리의 일부이다. 왼쪽 사각형의 윗 부분은 이미 존재하는 코드를 의미한다.

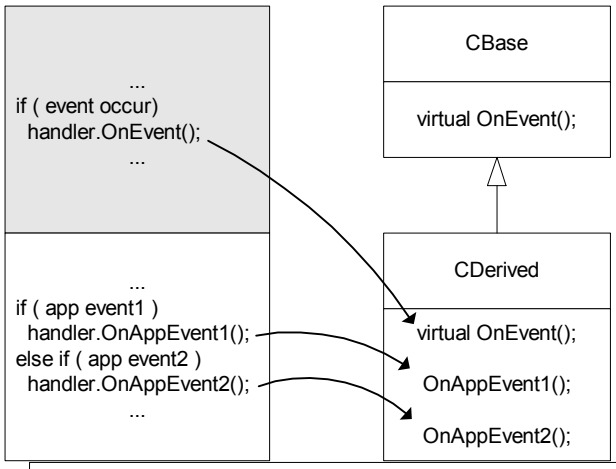
라이브러리의 소스 코드가 제공된다면 직접 CBase::OnEvent()를 수정할 수 있지만, 이것은 좋은 방법이 아니다. 독자들은 이미 해결책을 알고 있는데, 그것은 바로 가상 함수(virtual function)를 이용하는 것이다. 물론 CBase는 OnEvent()를 가상 함수로 가지고 있을 것이다.



[그림 5.16] 핸들러 호출의 런타임 바인딩: 가상 함수는 런타임 바인딩 되므로, 이벤트 발생은 추가된 새로운 이벤트 핸들러를 호출한다.

단지 CBase를 상속받아 CDerived를 구현하고, OnEvent()를 오버라이드(override)하는 것이다. 사실 가상 함수의 주된 목적은 바로 이러한 구현을 위해서이다.

생각해 볼 다른 문제는 이벤트 핸들러가 아니라 이벤트 자체에 대한 것이다. 이미 존재하는 코드에서 처리가 불가능한 이벤트는 후에 사용자에 의해 처리된다. 사용자 코드는 각 이벤트에 대응하는 이벤트 핸들러를 호출할 것이고, 처리할 이벤트 핸들러를 제공해 주어야 한다. 이 문제를 아래 [그림 5.17]처럼 해결할 수 있다.



[그림 5.17] 나중에 추가되는 이벤트 핸들러의 구현: 응용 프로그램에 의존적인 이벤트에 대해서는 라이브러리에서 자동화가 불가능하다. 왼쪽 그림의 아래쪽과 CDerived 클래스는 나중에 사용자가 추가한 코드를 의미한다.

이러한 이벤트 패턴의 문제점은 이벤트의 개수가 가변적이라는 것이다. 위 그림과 같은 구현은 이벤트가 늘어날수록 복잡한 if문이 얽혀져 있을 것이고, 유지/보수를 어렵게 한다. 이러한 이벤트 패턴의 문제점을 해결하기 위해 스테이트 패턴을 구현하기 위해 사용했던 방법을 사용할 수 있다.

예를 들면 키 입력에 대응해 이벤트 핸들러를 호출해야 하는 경우를 생각해 보자. 키 입력과 이벤트 핸들러를 다음과 같은 구조체로 표현할 수 있다.

```
struct StateMap
{
    int input;
    void (CAccount::*mfp)();
};
```

이제 CAccount 클래스는 StateMap의 배열을 가진다.

```
static StateMap map[];
```

그리고 이 배열은 다음과 같은 방법으로 초기화한다.

```
/*static*/ CAccount::StateMap CAccount::map[] =
{
    '1', CAccount::OnIdle,
    '2', CAccount::OnInput,
    '3', CAccount::OnWaitAck,
    0, NULL // sentinel
};//CAccount::map[]
```

이러한 초기화는 다음과 같은 이점이 있다.

1. 이벤트 입력과 이벤트 핸들러가 같이 표현되어 있어서, 이벤트의 추가가 쉽다.
2. 가변적인 개수의 이벤트를 다룰 수 있다. 마지막에 끝을 표시하는 sentinel에 주목하라.
3. 이벤트를 호출하는 부분에 대한 수정이 필요 없다. 그러므로 이벤트를 호출하는 코드는 라이브러리에 포함되어 사용자로부터 은닉되며, 사용자는 이벤트 핸들러에만 집중하여 코딩한다.

이벤트를 호출하는 코드는 다음과 같은 모양이 될 것이다.

```
        i  = 0;
        while ( CAccount::map[i].input != 0 )
        {
            if ( ch == CAccount::map[i].input )
            {
                (account.*(CAccount::map[i].mfp))();
            }//if
            ++i;
        }//while
```

그러므로 이벤트 패턴을 지원하는 라이브러리를 작성할 때, 최종 사용자가

이벤트 호출 메커니즘과 상관없이 이벤트에 집중해서 코딩하도록 할 수 있다. 아래의 예를 보자.

아래의 [예제 5.6]은 사용자가 서버를 선택하고, 계정을 입력하고, 서버로부터 응답을 기다리는 상태를 키 입력에 대한 이벤트로 처리해 보았다. 예를 들어, '3'을 누르면 서버로부터 응답이 돌아온 이벤트로 가정하여, 해당 이벤트 핸들러를 호출한다.

[예제 5.6] 이벤트 패턴의 구현

```
#include <iostream>
#include <conio.h>

#define KEY_ESC     27

using std::cout;
using std::endl;

class CAccount
{
public:
    struct StateMap
    {
        int input;
        void (CAccount::*mfp)();
    };
    static StateMap map[];

    enum
    {
        STATE_IDLE,
        STATE_INPUT,
        STATE_WAIT_ACK
    };

public:
    int     m_iState;

    CAccount();
    void OnIdle();
    void OnInput();
    void OnWaitAck();
```

```
};//class CAccount

CAccount::CAccount()
{
    m_iState = STATE_IDLE;
}

void CAccount::OnIdle()
{
    cout << "OnIdle()" << endl;
}

void CAccount::OnInput()
{
    cout << "OnInput()" << endl;
}

void CAccount::OnWaitAck()
{
    cout << "OnWaitAck()" << endl;
}

/*static*/ CAccount::StateMap CAccount::map[] =
{
    '1', CAccount::OnIdle,
    '2', CAccount::OnInput,
    '3', CAccount::OnWaitAck,
    0, NULL // sentinel
};//CAccount::map[]

void main()
{
    int         ch = 0;
    int         i;
    CAccount    account;

    while ( ch != KEY_ESC )
    {
        ch = getch();
        i  = 0;
        while ( CAccount::map[i].input != 0 )
        {
            if ( ch == CAccount::map[i].input )
```

```
            {
                (account.*(CAccount::map[i].mfp))();
            }//if
            ++i;
        }//while
    }//while
}//main()
```

키를 누르면 각 이벤트 핸들러를 호출한다. Esc를 누르면 프로그램은 종료한다.

우리는 이벤트 패턴을 6장에서 다시 살펴볼 것이다. 이벤트 패턴과 가상 패턴은 MFC 메시지 맵의 핵심이다!

싱글톤(singleton) 패턴

<절도비라>

이 절에서는 대중적으로 널리 사용되는 생성 패턴인 싱글톤의 개념과, 싱글톤을 사용하는 이점에 대해서 살펴본다. 또한 MFC에서 사용자가 최종적으로 생성하는 어플리케이션 객체의 이름을 모르고도 객체를 참조하는 코드를 사용하는 것이 싱글톤 패턴에 의해서 가능함을 이해한다.

</절도비라>

마지막으로 **싱글톤(singleton) 패턴**을 살펴보자. MFC의 전역 함수 중 AfxGetApp()는 사용자 코드가 생성한 CWinApp 타입의 객체의 주소를 리턴한다. 어떻게 나중에 구현될 객체의 주소를 얻는 것이 가능한가? 해결책은 바로 싱글톤 패턴을 사용하는 것이다.

싱글톤이란 어떤 타입의 객체가 프로그램의 전 영역에 걸쳐 단 하나만 존재하는 것을 말한다. 하지만 싱글톤의 이름과는 다르게 싱글톤을 사용하는 이유는 '단 하나'의 이유 때문이 아니라, 객체의 이름을 모르고도 객체의 함수를 호출하는 능력 때문이다.

예를 들면 CWinApp 클래스가 싱글톤인 것을 알면, 독자들은 후에 추가될 CWinApp 클래스 타입의 객체의 이름과는 상관없이, 기능을 미리 구현하는 것이 가능하다!

싱글톤의 핵심은 정적 변수를 이용하는 것인데, CWinApp의 경우, 클래스의 이름이 고정되어 있으므로, 간단하게 정적 변수를 추가하는 것만으로 싱글톤

의 구현이 가능하다. 하지만 조금 더 유연하게 싱글톤을 구현하여, 어떠한 클래스든지 싱글톤으로 만들 수 있도록 하자.

싱글톤을 구현하기 위해 아래 [예제 5.7]과 같이 정적 변수를 하나 가지는 템플릿 클래스를 만들 수 있다.

[예제 5.7] class KSingleton<>

```
template<typename T>
class KSingleton
{
private:
    static T*   ms_pSingleton;

public:
    KSingleton()
    {
        assert( NULL == KSingleton<T>::ms_pSingleton );
        KSingleton<T>::ms_pSingleton = (T*)this;
    }//KSingleton()

    ~KSingleton()
    {
        KSingleton<T>::ms_pSingleton = NULL;
    }//~KSingleton()

    static T* GetSingleton()
    {
        return ms_pSingleton;
    }//GetSingleton()
};//class KSingleton

/*static*/ template<typename T> T* KSingleton<T>::ms_pSingleton = NULL;
```

KSingleton 생성자는 먼저 정적 변수 ms_pSingleton이 NULL임을 검사한다. 이것은 KSingleton을 상속 받은 클래스의 객체가 두개 이상 만들어지는 경우, 프로그램을 종료하도록 한다. assert검사가 통과하면, this를 정적 변수에 대입한다. 즉 생성된 객체의 시작 주소를 정적 변수에 저장하는 것이다.

파괴자는 정적 변수를 다시 NULL로 초기화하고, 유일한 단 하나나의 정적 멤버 함수 GetSingleton()은 정적 변수를 리턴한다.

생성자의 동작은 이 클래스를 상속 받은 객체가 단 하나뿐임을 보장하고,

템플릿 구현은 어떠한 클래스든지 싱글톤으로 만들 수 있음을 의미하며, GetSingleton() 멤버 함수는 객체의 이름과 상관없이 객체를 접근하는 코드를 작성할 수 있음을 의미한다.

이제 CWinApp를 KSingleton<>에서 상속 받아 구현하여 [예제 5.8]과 같이 싱글톤으로 만들 수 있다.

[예제 5.8] CWinApp를 싱글톤으로 만들기

```
class CWinApp : public KSingleton<CWinApp>
{
public:
    void Print()
    {
        printf( "hello\n" );
    }//Print()
};//class CWinApp
```

KSingleton의 기능을 보여주는 예를 [예제 5.9]에 리스트하였다. main()에서 CWinApp 타입의 객체의 이름과 상관없이 함수를 호출하는 부분에 주목하라.

[예제 5.9] AfxGetApp()의 원리

```
#include <stdio.h>
#include <assert.h>
#include "vector"

#define AfxGetApp()   KSingleton<CWinApp>::GetSingleton()

template<typename T>
class KSingleton
{
private:
    static T*   ms_pSingleton;

public:
    KSingleton()
    {
        assert( NULL == KSingleton<T>::ms_pSingleton );
        KSingleton<T>::ms_pSingleton = (T*)this;
    }//KSingleton()
```

```
    ~KSingleton()
    {
        KSingleton<T>::ms_pSingleton = NULL;
    }//~KSingleton()

    static T* GetSingleton()
    {
        return ms_pSingleton;
    }//GetSingleton()
};//class KSingleton

/*static*/ template<typename T> T* KSingleton<T>::ms_pSingleton = NULL;


class CWinApp : public KSingleton<CWinApp>
{
public:
    void Print()
    {
        printf( "hello\n" );
    }//Print()
};//class CWinApp


CWinApp theApp;

void main()
{
    //KSingleton<CWinApp>::GetSingleton()->Print();
    AfxGetApp()->Print();
}//main()
```

출력 결과는 다음과 같다.

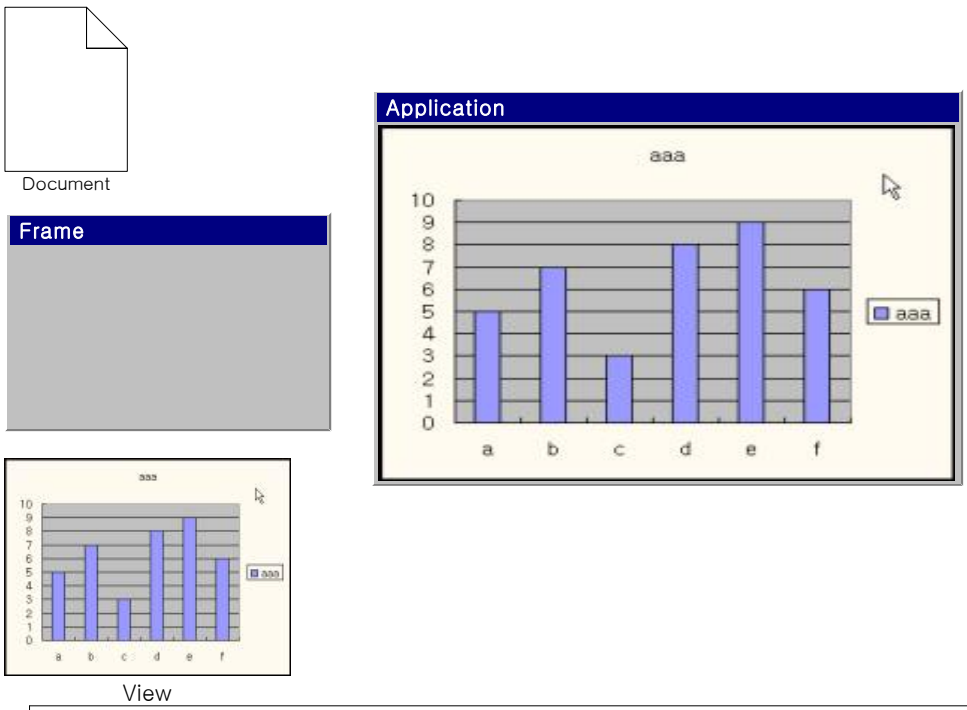
```
hello
```

AfeGetApp()의 실제 구현을 독자들이 꼭 확인해 보기 바란다.

요약

이벤트 패턴은 MFC의 핵심이다. MFC의 구조와 윈도우 구조에 의해 발생하는 이벤트에 대한 핸들러는 가상 함수로 구현되어 있다. 윈도우 메시지에 대응하는 이벤트 핸들러(메시지 핸들러)는 메시지 맵(message map) 테이블로 구현되어 있다.

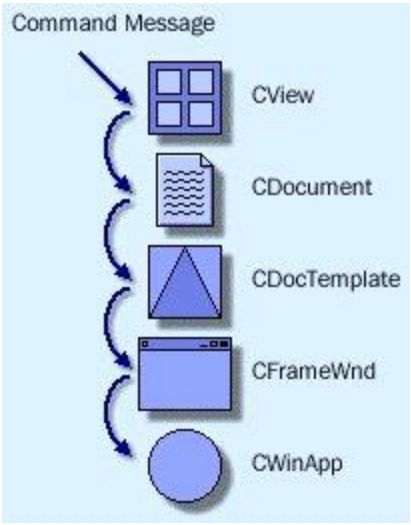
기 정의된 가상 함수 이벤트 핸들러의 종류와 이들의 호출 시점 및 상호 작용을 이해하는 것은 MFC 프로그래밍의 넘어야 할 산이다.



[그림 5.18] MFC의 다큐먼트/뷰 구조: MFC 싱글 다큐먼트 응용 프로그램의 경우, 응용 프로그램은 다큐먼트, 프레임, 뷰로 구성된다. 각각은 표현할 데이터, 응용 프로그램 윈도우, 클라이언트에 그리는 방식을 결정한다.

MFC의 다큐먼트/뷰 구조는 구조적으로는 브리지 패턴, 행위적으로는 관찰자 패턴이다. 우리는 표현해야 할 데이터와 이들의 렌더링을 구분해서 구현한

다. 또한 다큐먼트/뷰 구조는 윈도우 메시지 이벤트와도 관계되어 있다. 우리는 특정한 이벤트가 다큐먼트 혹은 뷰와 연관이 있는지 판단한 다음, 연관된 클래스에 윈도우 메시지 이벤트 핸들러를 작성한다.



[그림 5.19] MFC에서 메시지 핸들러의 검사 순서: 같은 이벤트 핸들러를 뷰와 다큐먼트에 모두 맵한 경우, 뷰의 핸들러를 먼저 검사한다.

MFC를 이루는 큰 다른 하나는 RTTI(run time type information)□와 관계

<저자 한마디>

MFC의 RTTI는 C++의 RTTI와 다르다. MFC의 초기 설계자들은 C++의 RTTI가 정해지기 전에 동적 생성과 실행시 타입 식별에 관한 것을 자신들의 라이브러리에 포함시킬 필요가 있었다.

</저자 한마디>

된 것이다. MFC의 RTTI는 10장에서 다룬다.

- **디자인 패턴**은 일반적 설계문제를 해결하기 위해 상호 교류하는 객체와 클래스에 대한 설명이다.
- **브리지 패턴**은 구현과 추상화 개념을 분리한다. MFC의 직렬화는 브리지 패턴을 사용한다.
- **관찰자 패턴**은 하나의 주제(subject)와 연관된 여러 관찰자(observer)의 동

작을 표현하는 행위 패턴이다. MFC의 다큐먼트/뷰는 관찰자 패턴을 사용한다.

- **스테이트 패턴**은 객체의 내부 상태에 따라 객체의 행동을 변경해야 하는 경우에 사용한다. 스테이트 패턴을 이벤트 처리에 적용할 수 있는데, MFC의 메시지 맵은 이벤트 패턴을 사용한다.
- **싱글톤 패턴**은 프로그램의 전체에 어떤 객체가 단 하나만 생성됨을 보장할 뿐만 아니라, 생성될 객체의 이름과 상관없이 인터페이스를 구현하는 것을 가능하게 한다.

[문서의 끝]