정적 다형성과 동적 다형성

객체 지향 프로그래밍의 특성을 세 가지로 요약하라고 한다면 아마도 '상속, 다형성, 그리고 캡슐화'를 꼽을 수 있을 것 같습니다. 적어도 제가 알고 있는 모든 객체 지향 언어는 위의 세 가지 특성을 기본적으로 지원하고 있으며 디자인 패턴과 같은 객체 지향 설계 전체를 꿰뚫는 핵심이 바로 이 세 가지 특성을 토대로 이루어 지기 때문입니다. 이전에 '상속과 합성 그리고 다중 상속'이라는 글에서 저는 상속과 캡슐화에 대해서 이야기를 했었습니다. 그리고 이번에는 다형성에 대해서 이야기를 해볼까 합니다.

다형성(polymorphism)이란 사전적으로는 문자 그대로 '다양한 형태를 지닌 성질'을 의미합니다. 그리고 객체 지향 프로그래밍에서 다형성이란 '외형적으로 동일한 심볼을 지녔으나 내부적으로 다른 구현을 가질 수 있는 특성'을 의미합니다.

다형성은 크게 정적 다형성과 동적 다형성으로 나눌 수 있습니다. 전에 제가 static 키워드에 대한 글에서 정적(static)과 동적(dynamic)이라는 단어가 프로그래밍 언어에서 가지는 의미에 대해 언급한 적이 있습니다. 여기서도 비슷한 의미의 해석이 가능합니다. 정적 다형성은 컴파일 타임 바인딩을 의미하며 동적 다형성은 런타임 바인딩을 의미합니다. 다시 말하자면 다형적인 특성이 정적 다형성에서는 컴파일러의 심볼 처리 매카니즘에 의해 이루어 지게 되며 동적 다형성에서는 실행 시간에 처리되는 – C++에서는 가상 테이블이라는 방법을 사용하여 – 별도의 매카니즘을 가집니다.

보통 객체 지향 언어에서 정적 다형성을 overloading이라고 하며 동적 다형성을 overriding이라고 부릅니다.(overloading, overriding이라는 용어는 다형성의 매카니즘 보다는 단순한 외형적인 특징을 표현하는 단어이기 때문에 개인적으로는 이런 용어가 마음에 들지 않습니다만 어쨌든…)

정적 다형성(혹은 overloading)은 동일한 함수 이름을 가지더라도 해당 함수가 가지는 파라미터들의 개수, 타입, 순서가 다를 경우 컴파일러가 다른 함수로써 인식 하는 것을 의미합니다. 예를 들어

```
void Func(int x);
void Func(double x);
```

위의 두 함수는 컴파일러에 의해 서로 다른 함수로써 해석이 됩니다. 물론

```
void Func(int x, int y);
```

이것 역시 다른 함수입니다. 뭐 이런 정도는 객체 지향 서적의 입문서 chapter 1에 해당하는 내용들이라 굳이 자세한 설명을 하지는 않겠습니다. 어쨌든 이런 함수 overloading을 이용하게 되면 컴파일러가 해당 함수 이름(심볼)을 해석하는 과정에서 – 컴파일 시간에 – 다른 함수들로 분류를 해주기 때문에 '정적 다형성' 이라고 부릅니다.

반면에 동적 다형성(overriding)은 동일한 함수 이름과 파라미터 특성을 지닌, 상속 관계에 있는 클래스들의 멤버 함수에 대해서 외형적으로 호출되는 타입에 상관없이 실제 생성된 객체의 함수가 호출되도록 처리 되는 특성을 의미합니다. 예를 들어,

```
class Base
    {
    public:
        ...
        virtual void Func(int x);
    }

class Derived : public Base
    {
    public:
        ...
        void Func(int x);
    }
```

이렇게 정의를 하면

```
Base* p = new Derived;
p->Func();
```

이 때 p->Func()구문은 외형상으로 Base 클래스의 함수를 호출한 것처럼 보이지만 실제로 Derived클래스의 Func()함수가 호출됩니다. 뭐 이것 역시 대략 chapter 3(이 특성에 대해 설명하려면 적어도 chapter 2정도에서 상속에 대한 설명이 있어야 할테니까...)정도에 해당하는 내용이라 잘 알고 계시리라 믿겠습니다. 어쨌든 다형성의 이런 특성에 의해 객체 지향 프로그래밍에서는 '동일한 인터페이스의 여러 객체, 동일한 객체의 여러 인터페이스'를 갖는 것이 가능합니다.

그런데 사실 제가 정말 말하고 싶은 것은 (상당수의 객체 지향 관련 서적에서 이미 잘 나와 있는) 다형성이 어떤 것이냐? 가 아니라 (실제 정말 중요하지만 그 어디에도 친절하게 설명되어 있지 않은) 이것을 언제 어떻게 사용해야 할 것인가? 입니다. 그리고 이제부터 이에 대해서 한번 제 나름대로의 시각을 가지고 이야기를 시작해 보겠습니다.

함수 분기

프로그래밍을 하다 보면 어떤 조건에 의해 여러 함수(혹은 루틴)중 하나를 선택하여 실행하여야 하는 상황이 자주 발생합니다. 예를 들어, 아래와 비슷한 형식의 구문을 작성해야 하는 경우가 생깁니다.

```
switch (type)
    {
    case Something1: DoSomething1(); break;
    case Something2: DoSomething2(); break;
    case Something3: DoSomething3(); break;
    case Something4: DoSomething4(); break;
    ...
    }
```

이렇게 적절한 조건에 의해 적절한 함수를 호출해야 하는 구문을 여기서는 함수 분기라고 부르겠습니다.(정식 용어는 아닙니다.)

대개의 경우 이런 함수 구문이 필요한 상황은 크게 아래와 같은 두 가지 경우 중 하나가 됩니다.

1. 중복 제거를 위해
2. 어떤 것을 실행해야 할지 프로그램 수행 중에 결정해야 할 때

좋은 프로그래밍을 위해 권장되고 있는 여러 가지 코딩 지침 중에 '한번만 작성하기'라는 것이 있습니다. 쉽게 말해서 중복되는 구문을 여러 군데에 코딩하지 말라는 뜻입니다. 예를 들어 프로그램 소스 곳곳에서 파일을 열어서 해당 파일의 사이즈를 구하는 기능을 필요로 한다면 필요시 마다 매번

```
DWORD fileSize = 0;
HANDLE file = CreateFile(...)
if (file != INVALID_HANDLE)
    {
    fileSize = GetFileSize(file, NULL);
    CloseHandle(file);
    }
```

이런 식으로 할 것이 아니라 아래처럼 해당 구문을 함수로 만들어서

```
DWORD GetFileSize(const char* fileName);
```

```
{
ASSERT(fileName);
DWORD fileSize = -1;
HANDLE file = CreateFile(fileName,...);
if (file != INVALID_HANDLE)
    {
    fileSize = GetFileSize(file, NULL);
    CloseHandle(file);
    }

return fileSize;
}
```

필요 시 마다 이 함수를 호출하게 만드는 것입니다. 이렇게 되면 해당 기능의 변경이나 에러 수정, 디버깅 시 훨씬 효율적인 처리가 가능합니다. 물론 가독성 면에서도 전체 소스 길이가 짧아지고 집중도가 생겨서 좋습니다.

그러나 세상 만사가 이렇게 단순하지만은 않은지라 중복된 부분이 반드시 모든 면에서 일치하리라는 법은 없습니다. 즉, 전체 루틴이나 알고리즘에서 많은 부분은 유사하지만 일부를 다르게 사용하고 싶은 경우가 있을 것입니다. 이런 경우 가변적인 부분을 아래처럼 수행할 수 있습니다.

```
void DoSomthing(int flag)
    {
    ... /// 공통된 부분 처리
    switch (flag)       /// 가변적인 부분 처리
        {
        case CASE_ONE: DoSomething1() break;
        case CASE_TWO: DoSomething2() break;
        case CASE_THREE: DoSomething3() break;
        ...
        }
    ... /// 나머지 공통된 부분 처리
    }
```

이런 식으로 작성하고 관련 처리가 필요할 때마다 DoSomething()함수의 flag값만 바꿔주면서 호출을 하게 되면 중복된 구현을 할 필요 없이 모듈화된 프로그래밍이 가능하게 됩니다. 이

렇게 프로그램 작성 시 가변 처리 부분의 어떤 동작을 수행할 지 해당 분기 조건이 결정되는 것들을 앞으로 **정적 분기**(컴파일 시간에 분기 조건이 결정됨)라고 부르겠습니다.

한편, 이와는 달리 프로그램 실행 도중에 사용자의 입력 값에 따라 어떤 동작을 적절히 수행해 줘야 하는 경우가 있을 수 있습니다. 예를 들어 알집이나 빵집과 같은 파일 압축 프로그램들은 보통 압축 파일을 풀 때 '현재 디렉토리에 풀기', '압축 파일 이름의 폴더를 생성하여 풀기', '압축을 풀 폴더 선택하기' 등의 옵션을 제공합니다. 그리고 그런 옵션들 중 사용자가 선택한 옵션에 따라 적절한 개별적인 기능을 수행한 후 압축을 풀기 시작합니다.

이런 경우 사용자가 선택한 옵션에 따른 사전 동작을 수행하는 루틴을 먼저 분기 수행하고 나머지 공통된 동작(압축 풀기)는 공통으로 수행하도록 구현을 하는 것이 적절할 것입니다. 이렇게 실행 시간에 분기 조건이 결정되는 것들을 앞으로 **동적 분기**라고 부르겠습니다.(정적 분기, 동적 분기라는 용어 역시 정식 용어가 아닙니다.)

정적 분기

위에서 잠시 언급했던 파일 사이즈를 구하는 함수를 다시 생각해 보겠습니다. 프로그래밍을 하다 보면 아마도 파일 사이즈 외에 기타 다른 정보들 – 파일 생성 날짜, 최종 수정 날짜, 사용권한 등등 – 또한 필요로 할 수 있습니다. 이런 경우,

```
DWORD GetFileSize(const char* fileName);
time_t GetFileCreateTime(const char* fileName);
time_t GetFileLastModificationTime(const char* fileName);
bool GetFileAuth(const char* fileName, int auth);
```

이런 식으로 각각의 기능마다 별도의 함수를 작성할 수도 있겠지만 이 경우 해당 파일의 핸들을 얻어오거나 에러를 처리하는 등의 구문은 계속 중복 구현되어야 할 것입니다. 따라서 이런 경우에는 하나의 인터페이스를 통해 중복된 기능을 처리하고 나머지 가변적인 부분은 적절하게 분기하여 처리해줄 수 있다면 더 좋을 것입니다. 그러나 이 경우 처리해야 할 각 함수의 파라미터가 다르기 때문에 일반적인 형식의 분기 구문을 이용하기가 무척 까다롭습니다.

만약 위에서 언급한 switch()문을 이용한다면 아래처럼 구현이 가능할 것입니다.

```
bool GetFileSize(HANDLE file, DWORD& size);
bool GetFileCreateTime(HANDLE file, time_t& createTime);
bool GetFileLastModificationTime(HANDLE file, time_t& modifTime);
bool GetFileAuth(HANDLE file, int auth);

bool GetFileInfo(const char* fileName, FileInfo& info, int flag)
    {
```

```
    ASSERT(fileName);
    bool ret = false;
    HANDLE file = CreateFile(fileName,...);
    if (file != INVALID_HANDLE)
        {
        switch (flag)
        {
        case FILE_SIZE:             ret = GetFileSize(file, info.fileSize);
        case FILE_CREATE_TIME:  ret = GetFileCreateTime(file, info.createTime);
        case    FILE_LAST_MODIFY:       ret    =    GetFileLastModificationTime(file,
    info.modifTime);
        ...
        }

    CloseHandle(file);
    }

    return ret;
  }
```

이제 중복된 부분은 적절히 처리 되었고 가변적인 부분은 적절히 모듈화가 되었습니다. 그러나 이런 경우 – 구조체를 사용함으로써 - 쓸데없는 메모리 오버헤드가 너무 크며 프로그래머가 매번 GetFileInfo()를 사용할 때마다 자신이 참조해야 할 적절한 구조체 멤버가 무엇인지 신경을 써줘야 할 부분이 너무 많습니다. 자칫 GetFileInfo(file, info, FILE_CREATE_TIME)를 호출하고나서 info.modifTime을 참조하는 실수를 할 경우 발생되는 에러는 디버깅하기가 생각외로 쉽지 않습니다.

물론 GetFileInfo(const char* filename, void* info, int flag); 이런 식으로 info 파라미터를 void 타입으로 하는 방법을 생각해 볼 수도 있습니다. 그러나 이런 void 타입을 사용하는 것은 타입 안정성을 떨어뜨리기 때문에 – 비록 메모리 오버헤드를 줄일 수 있을지는 모르겠지만 - 안정성 면에서 볼 때 위 경우보다 더 나쁘면 나빴지 결코 더 나은 선택이 못 됩니다. 이렇게 할 경우 프로그래머는 적절한 타입 변환에 대한 책임까지 져야 합니다. 혹시 '아니 세상에 바보같이 이렇게 함수를 만드는 사람이 어디 있나?' 하고 생각하시는 분들이 있을 지 모르겠습니다. 그러나 상당수의 Win 32 API 들이 위 두 부류와 유사한 구조를 가지고 있습니다.

이제 아마도 C++ 문법에 익숙하신 분들이라면 이런 상황에서 템플릿을 생각해 볼 수 있을 것입니다.

```
template <type name T>
bool GetFileInfo(const char* fileName, T& info, int flag)
    {
    ASSERT(fileName);
    bool ret = false;
    HANDLE file = CreateFile(fileName,...);
    if (file != INVALID_HANDLE)
        {
        switch (flag)
            {
            case FILE_SIZE:             ret = GetFileSize(file, info);
            case FILE_CREATE_TIME:      ret = GetFileCreateTime(file, info);
            case FILE_LAST_MODIFY:      ret = GetFileLastModificationTime(file, info);
            ...
            }
        CloseHandle(file);
        }

    return ret;
    }
```

이렇게 한다면 메모리에 대한 오버헤드는 사라지고 템플릿 사용에 의해 타입 안정성까지 가지게 됩니다. 그러나 여전히 switch문 자체가 가진 default 처리 문제가 고질적인 병폐로 남아 있습니다. switch 문은 프로그래머가 설정한 case외의 flag값이 들어오는 경우에 대한 예외 처리가 매우 취약합니다. 예를 들어 코딩상의 실수에 의해 flag값으로 잘못된 정수 값을 넘겨 줄 경우 이 함수에서 할 수 있는 일은 단지 default문에 의해 예외를 던지거나 false를 반환하는 것 뿐입니다. 그러나 이것은 그다지 바람직한 현상이 못 됩니다. 왜냐하면 여기서 에러의 원인은 프로그램 실행 상의 문제가 아닌 코딩상의 실수였고 따라서 그런 실수는 당연히 – 예외나 리턴 값과 같은 – 실행 시간 처리가 아니라 – 문법 오류나 링크 오류와 같은 – 컴파일 시간 처리에 의해 체크가 돼줘야 하기 때문입니다. 결국 우리가 이 상황에서 필요한 것은 '**분기 결정이 실행 시간이 아니라 컴파일 시간에 이루어 져야 하는 것**' 입니다. 따라서 이 경우 switch문과 같은 실행 시간 분기 구문은 적절한 선택이 되지 못합니다.(마찬가지의 이유로 if / else if 구문 역시 해결 방안이 되지 못합니다.)

이제 정적 다형성에 대해서 생각해 볼 때입니다. 정적 다형성은 함수의 파라미터 개수나 타입, 순서에 따라 컴파일 시간에 함수 분기가 이루어지는 매카니즘을 말합니다. 이것은 우리가 처한

현재 상황에 잘 맞습니다. 이제 정적 다형성을 이용하면 아래와 같이 GetFileInfo()함수를 수정할 수 있습니다.

```
bool GetFileInfo(HANDLE file, DWORD& size);          /// get file size
bool GetFileInfo(HANDLE file, time_t& createTime);   /// get file create time
bool GetFileInfo(HANDLE file, time_t& modifTime);    /// get file last modification time
bool GetFileInfo(HANDLE file, int auth);             /// get file authorization

template <type name T>
bool GetFileInfo(const char* fileName, T& info)
    {
    ASSERT(fileName);
    bool ret = false;
    HANDLE file = CreateFile(fileName,...);
    if (file != INVALID_HANDLE)
        {
        ret = GetFileInfo(file, info);
        CloseHandle(file);
        }

    return ret;
    }
```

이렇게 하면 템플릿 인자로 넘어오는 타입 T에 따라 적절한 함수 분기가 컴파일 시간에 이루어 집니다. (예를 들어 info 타입이 DWORD이면 파일 사이즈를 구하는 GetFileInfo(HANDLE, DWORD&)함수가 호출될 것입니다.)

그러나 여기에는 한 가지 문제가 있는데 GetFileCreateTime()과 GetFileLastModificationTime()의 파라미터 값들이 동일하다는 데 있습니다. 이 경우 정적 다형성에서는 두 함수를 구분할 수 있는 방법이 없기 때문에 컴파일러는 함수 호출 구문이 모호하다는 에러를 발생할 것입니다. 따라서 우리는 이러한 동일한 파라미터를 가진 함수들을 구분해 줄 수 있는 어떤 매카니즘이 필요합니다. 다시 말하면 flag값이 해주던 역할을 함수 오버로딩 시점에서 처리해 줄 수 있는 어떤 요소 즉, flag'값'이 아닌 flag'타입'을 만들어 줄 수 있다면 이러한 문제가 세련되게 처리될 것입니다.

Andrei Alexandrescue는 'Modern C++ Design'이라는 책에서 이런 역할을 수행할 수 있는 간단한 클래스 템플릿을 소개하고 있습니다. 이 클래스 템플릿은 어떤 상수 값을 하나의 타

입으로 만들어 주는 마법과 같은 역할을 수행합니다. 이 클래스 템플릿은 놀랍게도 아래와 같은 매우 단순한 구조를 가지고 있습니다.

```
template <int N>
struct Int2Type
    {
    enum { value = N };
    };
```

이제 이것을 아래와 같이 사용하면 됩니다.

```
Int2Type<1> One;
Int2Type<2> Two;
Int2Type<3> Three;
...
```

이제 생성된 각각의 객체들은 서로 별개의 클래스 타입을 가집니다. 다시 말하면 Int2Type<1>과 Int2Type<2>는 완전히 다른 클래스(객체가 아닙니다!)입니다. 따라서 이 클래스를 이용하면 상수 값을 이용하여 함수 오버로딩에 의한 분기를 처리할 수 있습니다. 즉,

```
void Func(Int2Type<1>);
void Func(Int2Type<2>);
```

이 두 함수는 완전히 다른 함수입니다. 이제 우리는 정적 다형성을 이용한 함수 분기 작업을 아래와 같이 수행할 수 있습니다.

```
enum { FILE_SIZE, FILE_CREATE_TIME, FILE_LAST_MODIFY, FILE_AUTH }

bool GetFileInfo(HANDLE file, DWORD& fileSize, Int2Type<FILE_SIZE>);
bool GetFileInfo(HANDLE file, time_t& createTime, Int2Type<FILE_CREATE_TIME>);
bool GetFileInfo(HANDLE file, time_t& modifTime, Int2Type<FILE_LAST_MODIFY>);
bool GetFileInfo(HANDLE file, int& auth, Int2Type<FILE_AUTH>);

template <int Flag, typename T>
Bool GetFileInfo(const char* fileName, T& info)
    {
```

```
    ASSERT(fileName);
    bool ret = false;
    HANDLE file = CreateFile(fileName,...);
    if (file != INVALID_HANDLE)
        {
        ret = GetFileInfo(file, info, Int2Type<Flag>);
        CloseHandle(file);
        }

    return ret;
    }
```

위의 함수는 아래와 같이 사용하면 됩니다.

```
DWORD fileSize;
time_t createTime, lastModifTime;

GetFileInfo<FILE_SIZE>("sample.txt", fileSize);
GetFileInfo<FILE_CREATE_TIME>("sample.txt", createTime);
GetFileInfo<FILE_LAST_MODIFY>("sample.txt", lastModifTime);
```

이제 모든 분기 작업은 컴파일러의 템플릿 인스턴스화 및 함수 오버로딩 매카니즘에 의해 정적이고 세련되게 처리될 것입니다. 타입 안정성은 템플릿이 적절하게 책임져 줄 것이며 잘못된 flag값은 컴파일러의 정적 함수 바인딩 과정에서 적절하게 걸러질 것입니다. (만약 프로그래머의 실수에 의해 정의되지 않은 flag값을 사용할 경우 – 실행 시점이 아닌 – 컴파일 시점에서 정의되지 않은 함수 호출에 대한 에러가 발생할 것입니다.) 게다가 우리는 별도의 flag 값 비교/처리 과정에 대한 실행 시간 오버헤드 역시 제거할 수 있습니다. 심지어 각 함수 호출 구문은 어떤 처리를 하는 지가 템플릿 인자로 명확하게 드러나기 때문에 가독성 면에서도 우수합니다. (코드의 주석화란 바로 이런 것을 말하는 게 아닐까요?)

한 가지 아쉬운 점은 VC++6.0에서는 위와 같은 함수 템플릿이 잘 동작하지 않는다는 점입니다. 보다 정확하게 말하면 일반 함수 템플릿은 잘 동작하지만 위 함수 템플릿이 어떤 클래스의 멤버로써 사용 되면 컴파일 에러가 발생합니다. VC++6.0은 멤버 함수의 템플릿 인자 처리가 제대로 되지 않기 때문입니다. (테스트를 해보지는 않았지만 VC++7.x 버전에서는 잘 동작할 것입니다.)

어쨌든 여기서 중요한 것은 정적인 처리가 필요한 구문은 동적 매카니즘으로 구현했을 때 보다 정적 다형성이나 템플릿과 같은 정적 매카니즘을 이용하여 구현할 때 보다 깔끔하게 해결될 수 있다 라는 사실입니다.

동적 분기

이번에는 위에서 예로 들었던 압축 프로그램에 대해서 이야기를 해보도록 하겠습니다. 우리가 개발해야 할 압축 프로그램은 압축 풀기 옵션으로 위에서 언급했던 세 가지 방법('현재 디렉토리에 풀기', '압축 파일 이름의 폴더를 생성하여 풀기', '압축을 풀 폴더 선택')을 제공합니다. 이 때 분기 조건은 실행 시간에 결정되는 사항이기 때문에 정적 분기 방법을 사용할 수는 없습니다. 따라서 동적 분기 매카니즘을 활용해야 합니다. 우선 가장 쉽게 생각해 볼 수 있는 것이 switch문을 이용하는 것입니다.

```
void Extract(const char* fileName)
    {
    int option = GetUserOption();/// 사용자가 선택한 옵션 값을 얻어온다.
    std::string dirName;
    switch (option)
        {
        case CURRENT_DIRECTORY:
            dirName = GetCurrentDirectory();
            break;
        case USE_EXTRACT_FILENAME:
            dirName = filename;
            CreateDirectory(fileName);
            break;
        case USER_SELECTION:
            dirName = GetDirName();
            CreateDirectory(dirName.c_str());
            break;
        default:
            throw exception("invalid option value!!!");
        }

    …  // 압축 풀기 과정 구현
    }
```

switch문은 다중 분기를 처리하는데 있어 단순하고 쉬우며 성능 역시 뛰어난 방법입니다. 위의 경우 switch문을 사용하는 것은 좋은 선택이 될 수 있습니다. 그러나 switch문은 아래와 같은 단점들이 있습니다.

1. 분기 조건으로 정수형 상수만 사용이 가능하다.
   - 정수 상수만을 조건식에 사용할 수 있다는 점은 switch문의 가장 큰 단점입니다. 때로는 사용자의 입력 스트링 값에 따른 실행 시간 분기가 필요할 수 있으며 어떤 리소스 핸들 값에 의한 분기 역시 가능한 상황입니다. 그러나 switch문은 이러한 것들을 전혀 해결해 주지 못합니다. 심지어 미리 저장된 어떤 변수 값들에 의한 분기 역시 불가능합니다.
2. 분기 구문의 확장 및 수정이 용이하지 못하다.
   - 추가적인 분기 구문이 필요할 때 마다 switch문은 점점 커지게 됩니다. 이것은 밑에서 언급할 가독성 면에서 결코 좋지 않습니다. 게다가 그보다 더 중요한 사실은 switch문이 각 분기 조건에서 필요로 하는 클래스나 소스 파일들을 모두 참조해야 하는 컴파일 의존성을 가지고 있다는 사실입니다. 이것은 유지 보수 면에서 결코 좋지 못한 특성입니다.
3. 가독성이 떨어진다.
   - 분기 구문이 10개 정도라면 괜찮지만 수십 개, 수백 개에 이른다면 switch문은 함수의 덩치를 키우는 주범이 됩니다. 게다가 많은 분기 구문들 중 특정 조건 구문을 찾아 유지 보수나 디버깅을 한다는 것은 무척 고된 작업이 될 것입니다.(100 개 정도 되는 case문을 가진 switch 구문은 보고만 있어도 난시를 유발할 지 모릅니다.)

우선 1번의 단점을 극복하기 위해 선택할 수 있는 것은 if / else if 문을 사용하는 것입니다. 조건식이 몇 개 되지 않는 간단한 분기 구문이라면 if / else if를 사용하는 것이 가장 좋을 것입니다. 그러나 if / else if는 다중 분기 사용 시 가독성이나 유지 보수성 면에서 switch문보다 훨씬 나쁩니다. 게다가 만약 분기 조건식이 복잡하고 조건 식의 개수가 굉장히 많은 상황이라면 if / else if는 실행 시간 성능에 까지 나쁜 영향을 미칩니다. 따라서 if / else if는 switch를 대체할 만한 방안이 되지 못합니다.

만약 C에서 프로그래밍을 많이 해보신 분이나 MFC의 내부 구조를 공부 해본 분이라면 함수 포인터 테이블을 생각할 지 모르겠습니다. 위의 Extract()함수를 함수 포인터 배열에 의한 분기 구문으로 바꾸면 아래와 같습니다.

```
typedef const char* (*fnType)(const char*);

const char* CurDirDispatch(const char* fileName)
    {
```

```
    return GetCurrentDirectory();
    }

const char* ExtFileNameDispatch(const char* fileName)
    {
    CreateDirectory(fileName);
    return fileName;
    }

const char* UserSelectDispatch(const char* fileName)
    {
    std::string temp = GetDirName();
    CreateDirectory(temp.c_str());
    return temp.c_str();
    }

std::vector<fnType> dispatchFun;

void RegisterFunction(fnType fn)/// 함수 등록 시 사용
    {
    dispatchFun.push_back(fn);
    }

/// 분기 함수 등록
RegisterFunction(CurDirDispatch);
RegisterFunction(ExtFileNameDispatch);
RegisterFunction(UserSelectDispatch);

void Extract(const char* fileName)
    {
    std::string dirName;
    try {
        dirName = dispatchFun.at(GetUserOption())(fileName);
        }
    catch (out_of_range& e) /// at()은 유효값을 벗어나면 out_of_range 예외를 발생시킴
        {
```

```
        throw exception("invalid option value!!!");
        }
    ...   /// 압축 풀기 과정 구현
    }
```

switch문에 비해 다소 복잡해 보입니다만 이런 함수 포인터 집합을 이용한 방식은 분기 구문이 많아 졌을 때 제 값을 하게 됩니다. 즉, 분기 구문이 많아 지더라도 기존 구현 부분은 전혀 영향을 받지 않으며 단지 새로운 분기 조건을 처리할 함수들만을 추가해주면 되기 때문입니다. 게다가 Extract()함수는 각 분기 모듈과의 의존 관계가 없기 때문에 유지 보수 면에서도 효과적입니다. 물론 분기 규모가 큰 경우 가독성 면에서도 switch문 보다 좋습니다. 따라서 switch문이 가졌던 2번, 3번 문제를 해결할 수 있습니다. 단, 이 경우 역시 1번의 단점은 해결할 수 없습니다. vector나 배열은 정수 값만을 인덱스로 가질 수 있기 때문입니다.

그러나 C++라면 1번의 단점 역시 해결할 수 있습니다. C++에서는 map이라고 하는 아주 멋진 컨테이너를 표준 라이브러리로 지원하기 때문입니다. map은 인덱스 key로 - 해당 타입이 크기 비교 연산자를 지원하거나 적절하게 재정의 되어 있다면 - 어떤 타입이든지 가질 수 있습니다. 따라서 함수 포인터를 map의 value로 가지고 각 분기 조건을 위한 값을 key로 갖는다면 정수나 스트링 뿐만이 아니라 심지어 객체 자체를 분기 조건을 위한 key로 사용할 수 있습니다. 따라서 함수 포인터 배열이 갖는 장점을 가지면서 훨씬 유연한 처리가 가능합니다. 게다가 if / else if 보다 분기 조건 검색에 있어서 - 평균적으로 - 훨씬 좋은 성능을 보여줍니다. Extract() 함수에 map을 적용하면 아래와 같습니다.

```
typedef std::map<DWORD, fnType>   FunDispatchType;
...
void Extract(const char* fileName)
    {
    FunDispatchType::iterator pos = dispatchFun.find(GetUserOption());
    if (pos == dispatchFun.end())
        throw exception("invalid option value!!!");

    std::string dirName = pos->second(fileName);
    ... /// 압축 풀기 과정 구현
    }
```

참고로 MFC의 메시지 맵이 함수 테이블을 사용한 구조를 가지고 있습니다.
하지만 함수 포인터 테이블 이용 시에도 단점은 있습니다.

1. 분기 함수의 원형이 동일한 구조로 되어 있어야 합니다.
   - 함수 포인터를 저장하기 위해서는 저장할 함수들이 모두 동일한 타입(즉, 파라미터 개수, 타입, 순서, 리턴값)을 가져야 합니다. 이것은 분기 구문을 경직된 구조로 만드는 문제를 가집니다. 대부분의 경우 이 단점이 크게 문제되지 않지만 만약 유지 보수 단계에서 특정 분기 함수에 추가적인 파라미터를 집어 넣어야 하는 상황이 발생되면 전체 분기 함수들의 원형을 수정해줘야 할 것입니다.
2. 분기 함수들이 참조해야 할 변수의 숫자가 매우 많은 경우 가독성이나 코딩 면에서 불편함을 초래합니다.
   - 가령 각 분기 함수들이 메인 호출 함수에서 처리되는 변수들 10개를 참조해야 한다면, 이것은 각 분기 함수들의 파라미터 개수가 최소한 10개는 되어야 한다는 것을 뜻합니다. 이것은 코딩을 하는 입장에서나 나중에 유지 보수를 수행하는 사람의 입장에서나 썩 내키지 않는 상황일 것입니다.
3. 분기점이 여러 군데에 발생되는 루틴에 대한 처리에는 적절하지 않습니다.
   - 이것은 함수 테이블을 여러 개 사용하거나 혹은 분기 함수 내에서 또 다른 분기 처리를 해줘야 한다는 것을 의미합니다.
4. 바람직한 캡슐화 구조가 아닙니다.
   - 각 분기 함수들이 효과적으로 캡슐화 되었다고 보기 힘들며 만약 분기 함수에서 처리해야 할 동작이 매우 많아서 하나의 함수에서 다 처리하기 곤란한 경우 캡슐화는 쉽게 깨질 수 있습니다.

동적 다형성을 이용한 분기

자 이제 우리가 맨 처음 이야기를 시작했었던 동적 다형성에 대해서 생각해 봐야 할 시간이 왔습니다.

혹시 객체 지향 설계나 디자인 패턴 관련 공부를 해보신 분들은 'switch문을 다형성을 이용한 설계로 바꿔라' 라는 말을 들어 보셨을 것입니다. 이것은 아래와 같은 것을 말합니다.

```
class Base
    {
    public:
        virtual Something() = 0;
    }

class Sub1 : public Base
    {
    public:
        Something() { std::cout << "do something1...₩n" };
```

```
    }

class Sub2 : public Base
    {
    public:
        Something() { std::cout << "do something2...₩n" };
    }

class Sub3 : public Base
    {
    public:
        Something() { std::cout << "do something3...₩n" };
    }
```

이렇게 클래스 구조를 만들게 되면 우리는 아래의 구문을

```
void DoSomething(int flag)
    {
    switch (flag)
        {
        case 1: std::cout << "do something1...₩n";    break;
        case 2: std::cout << "do something2...₩n"; break;
        case 3: std::cout << "do something3...₩n"; break;
        }
    }
```

이렇게 바꿀 수 있습니다.

```
void DoSomething(Base& obj)
    {
    obj.Something();
    }
```

flag을 통해 함수 분기가 이루어 지던 이전 버전과 달리 동적 다형성을 이용하게 되면 어떤 객체를 생성하느냐에 따라 실행 시간 분기가 가능해 집니다. 이러한 동적 다형성을 이용한 구조는 매우 높은 유연성과 응집력, 낮은 결합도를 가집니다. DoSomething()함수는 실제 자신이 처리

해야 할 것이 무엇인지 전혀 알 필요가 없어지며 따라서 마치 함수 포인터 테이블을 이용한 것처럼 변화에 유연하게 됩니다. 반면 참조가 필요한 관련 파라미터를 객체 생성 시 미리 넘겨줄 수 있으므로 함수 호출 시에 필요한 파라미터 역시 유동적인 조절이 가능합니다. 각각의 객체별로 필요한 파라미터들은 생성 시 미리 멤버로써 저장을 하고 공통적으로 필요한 일부 파라미터만 호출 인터페이스를 통해서 받을 수도 있습니다.

그리고 분기에서 처리해야 할 일이 많아서 여러 함수의 호출이 필요할 때도 각각의 클래스 별 캡슐화가 가능하므로 관련 있는 함수끼리의 높은 응집력을 유지할 수 있습니다. 위의 압축 풀기 함수에서 압축 풀기는 각각의 압축 파일의 형식에 따라 독립적인 알고리즘 구조를 가지고 있을 것입니다. 그리고 각각의 압축 알고리즘을 수행하는 것은 아마도 그리 간단치 않을 것입니다. 유지해야 할 변수들의 양도 많을 것이며 수행해야 할 단계별 수행 동작들도 많을 것입니다. 이런 것을 switch문이나 함수 테이블을 이용하여 함수 호출 처리를 하는 것은 난해한 구조가 되기 쉽습니다. 이런 경우 각각의 압축 파일 별로 클래스를 만들고 이런 압축 알고리즘 클래스들을 공통의 인터페이스로 캡슐화한다면 훨씬 유연하고 바람직한 구조가 될 것입니다.

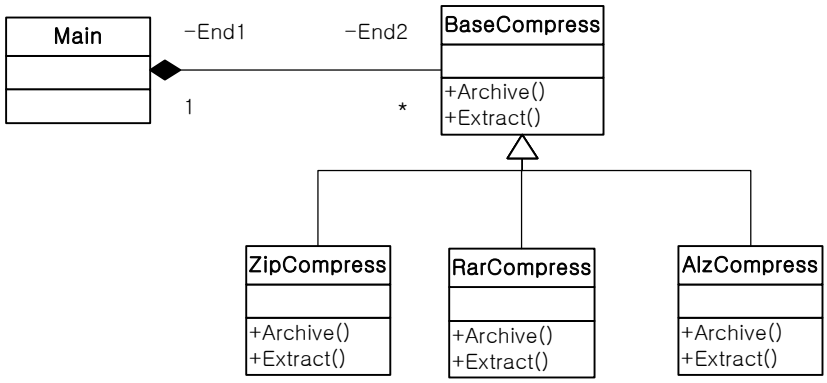
```
class BaseCompress
    {
    public:
        virtual bool Archive() = 0;
        virtual bool Extract() = 0;
    }

class ZipCompress : public BaseCompress
    {
    public:
        virtual bool Archive();
        virtual bool Extract();

    private:
        ...
    }

class RarCompress : public BaseCompress
    {
    public:
        virtual bool Archive();
        virtual bool Extract();
```

```
    private:
        ...
    }

class AlzCompress : public BaseCompress
    {
    public:
        virtual bool Archive();
        virtual bool Extract();

    private:
        ...
    }
```

이제 압축 풀기 함수는 아래처럼 구현될 수 있을 것입니다.

```
void Extract(const char* fileName)
    {
    ...

    BaseCompress* comp = CreateCompressObj(GetFileType(fileName));
    comp->Extract();
    }
```

압축 하기 함수 역시 이와 비슷하게 구현할 수 있습니다. 어쨌든 이제 우리는 압축에 관련된 기능들을 클래스로 캡슐화하여 응집력을 높일 수 있었고, 압축 파일 형식에 따라 별도의 상세 클래스들로 구현함으로써 각각의 알고리즘간의 결합도를 낮출 수 있었으며 분기가 필요한 부분을 동일한 인터페이스로 캡슐화함으로써 변화에 유연하게 대처할 수 있게 되었습니다. 이것을 클래스 다이어그램으로 표현한다면 아래와 같습니다.



그리고 위와 같은 구조를 디자인 패턴에서는 Strategy 패턴이라고 부릅니다. 즉, 각각의 가변적인 압축 알고리즘 정책을 캡슐화함으로써 공통된 부분의 재사용을 높일 수 있도록 유연성을 확보하는 구조입니다. 이것은 함수 테이블이 갖는 단점들을 보완할 수 있습니다.

1. 함수의 원형이 동일한 구조로 되어 있어야 합니다. -> 생성자 인자를 다르게 처리하거나 별도의 멤버 셋팅 함수를 사용함으로써 결점을 보완할 수 있습니다.
2. 분기 함수들이 참조해야 할 변수의 숫자가 매우 많은 경우 가독성이나 코딩 면에서 불편함을 초래합니다. -> 역시 클래스 단위로 처리되므로 보완이 가능합니다.
3. 분기점이 여러 군데에 발생되는 루틴에 대한 처리에는 적절하지 않습니다. -> 이것은 다음에 소개하는 다른 패턴을 이용하여 극복이 가능합니다.
4. 바람직한 캡슐화 구조가 아닙니다. -> 적절히 캡슐화된 클래스 구조를 이용함으로써 높은 응집력과 낮은 결합도를 가진 설계 구조를 가질 수 있습니다.

이제 여러 분기점을 가지는 루틴은 동적 다형성을 이용하여 어떻게 처리할 수 있을 지 생각해 보겠습니다. 사실 위의 구조를 완벽히 이해하였다면 이것 역시 쉽게 해결할 수 있습니다. 계속해서 압축 풀기 과정을 예로 들어 본다면, 각 압축 파일 별로 알고리즘이 다르더라도 – 저는 압축 알고리즘에 대해 잘 모릅니다만 – 전체 수행을 위한 work flow는 비슷할 수 있습니다. 예를 들어 압축 알고리즘은 보통 아래와 같은 흐름을 갖는다고 가정하겠습니다.(다시 한번 강조하지만 이 예는 실제와 다를 수 있습니다.)

1. 파일을 연다.
2. 해더 데이터를 읽는다.
3. 해더 정보를 개별적인 알고리즘 방식대로 해석한다.
4. 해석된 정보를 바탕으로 다음 위치에서부터 일정 사이즈만큼의 데이터를 읽는다.
5. 각각의 알고리즘에 의해 해당 데이터를 디코딩한다.
6. 4, 5번 동작을 파일 끝에 다다를 때 까지 반복 수행한다.

7. 파일 끝에 다다르면 무결성 체크를 한다.
8. 압축 풀기 동작 완료

만약 현재 구현하고자 하는 대부분의 압축 알고리즘이 위와 같은 흐름을 가지고 처리된다면 이런 흐름에 대한 구현은 BaseCompress 클래스에서 수행하고 개별적인 동작에 해당하는 부분만 따로 하위 클래스에서 수행하기 위한 가상함수로 처리할 수 있습니다. 즉, 아래와 같은 구현이 가능합니다.

```
class BaseCompress
    {
    public:
        bool Archive();
        bool Extract();   // 이제 가상함수가 아닙니다.

    protected:
        virtual void ReadHeader(vector<unsigned char>& hdr) = 0;
        virtual int AnalyzeHeader(const vector<unsigned char>& hdr) = 0;
        virtual bool ReadData(int size, vector<unsigned char>& hdr) = 0;
        virtual void UnCompress(const vector<unsigned cha*>& encData,
vector<unsigned char>& decData) = 0;
        virtual bool CheckIntegrity() = 0;
    }

bool BaseCompress::Extract()
    {
    vector<unsigned char> tmp, decBuf;
    ReadHeader(tmp);
    int readSize = AnalyzeHeader(tmp);
    while (ReadData(readSize, tmp))
        {
        UnCompress(tmp, decBuf);
        }

    return CheckIntegrity();
    }
```

이제 전체 동작 흐름은 BaseCompress가 구현하였기 때문에 상속받은 각각의 압축 알고리즘 클래스들에서는 해당 알고리즘 세부 동작(가상 함수 부분)만을 구현하면 됩니다. 즉, 모든 공통된 동작은 추상 클래스에 일임하고 가변적인 부분만 상세 클래스에서 구현하는 것입니다. 이렇게 하면 여러 분기점을 가진 문제가 동적 다형성에 의해 효과적으로 처리될 수 있습니다. 이러한 설계 구조를 디자인 패턴에서는 Template Method 패턴이라고 합니다. 외형적인 클래스 구조는 Strategy 패턴과 유사하지만 각각의 상세 클래스 들이 공통된 flow를 가지느냐 그렇지 않느냐의 차이가 있습니다.

실제 프로그램을 개발할 때 이런 다중 분기점 문제는 상당히 자주 발생되며 따라서 Template method 패턴 역시 매우 자주 사용되는 패턴입니다. (저 역시 실전에서 가장 자주 사용하는 패턴 중 하나입니다.)

그렇다면 동적 다형성을 이용하는 방법에는 어떤 문제가 있을 까요? 여기까지 봤을 때는 동적 다형성이야말로 우리의 모든 고민을 해결해 주는 완벽한 해결책처럼 보입니다. 그러나 아쉽게도 이런 동적 다형성을 이용한 함수 분기 패턴들은 근원적인 문제점(이라기 보다는 고민거리)를 우리에게 안겨줍니다.

그것은 바로 상세 클래스 객체를 어떻게 생성할 것인가? 라는 것입니다.

위의 예에서 우리는 각각의 분기 동작을 가상 함수 매카니즘을 이용한 동적 다형성을 통해 얻을 수 있었습니다. 하지만 정작 적절한 분기를 위한 알맞은 객체를 어디에서 어떻게 생성해야 할 것인가에 대해서는 미처 고민하지 못했습니다. 그리고 이점을 생각해 보면 결국 객체 생성에 있어서 새로운 다중 분기 문제를 갖게 됩니다.

```
void Extract(const char* fileName)
    {
    ...
    BaseCompress* comp = CreateCompressObj(GetFileType(fileName));  ///
CreateCompressObj()는 어떻게 구현할 것인가?
    comp->Extract();
    }
```

이것이 동적 다형성을 가진 구조에서 치러야 할 가장 비싼 비용입니다. 디자인 패턴을 조금이라도 공부하신 분들이라면 이것을 보고 바로 Factory 패턴을 생각할 수 있을 것입니다. (대부분의 디자인 패턴 책 앞부분을 장식하기 때문에) 그러나 Factory 패턴을 실제 구현하는 문제는 그리 녹녹한 문제가 아닙니다. Factory 패턴을 구현하는 것 또한 결국 또 다른 함수 분기 문제를 처리해야 함을 의미하기 때문입니다. 즉, 어떤 조건에서 어떤 객체를 생성하여 넘겨 줄 것인가 하는 문제가 있습니다. 따라서 앞서 계속 언급했던 다양한 해결책들이 Factory 패턴에 적용 가능합니다.

```
BaseCompress* CreateCompressObj(int type)
    {
    switch (type)
        {
        case ZIP_TYPE: return new ZipCompress;
        case RAR_TYPE: return new RarCompress;
        case ALZ_TYPE: return new AlzCompress;
        default:
            throw bad_alloc;
        }
    }
```

혹은 map을 이용하는 것도 가능합니다.

```
typedef BaseCompress* (*createFunType)();
typedef std::map<int, createFunType > FactoryType;
FactoryType objFactory;

void RegisterCreateObjFun(int type, createFunType  fn)
    {
    objFactory.insert(make_pair(type, fn));
    }

BaseCompress* CreateCompress(int type)
    {
    FactoryType::iterator pos = objFactory.find(type);
    if (pos == objFactory.end())
        throw bad_alloc;

    return pos->second();
    }
```

아마도 간단한 구현을 위해서는 switch문도 괜찮지만 낮은 결합도와 높은 유연성을 위해서는 – 그러나 구현에 따른 비용은 더 큰 – map을 이용한 방식이 더 좋을 것입니다. 혹은 정적 다형성을 이용하여 구현할 수도 있습니다. 심지어 Factory 자체도 동적 다형성을 이용할 수 있습니다. 모든 것은 상황에 맞는 선택의 문제일 뿐입니다. 중요한 것은 어떤 방식을 사용하더라도 외

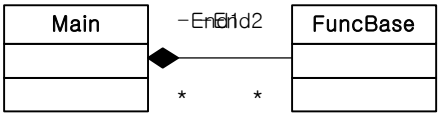
부로 노출되는 인터페이스(여기서는 CreateCompress() 메소드)는 변경되지 않아야 합니다. Factory 패턴을 사용하는 가장 중요한 이유는 바로 이것입니다. 여기서 우리는 새로운 사실을 알 수 있습니다.

기능의 확장과 기능의 추가

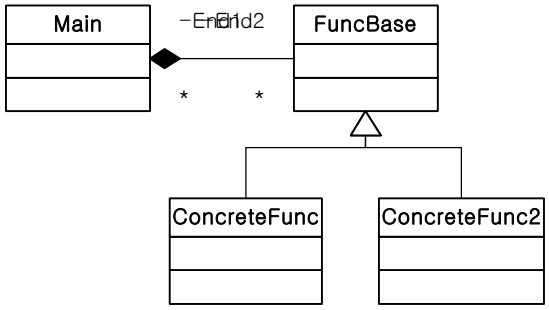
기능을 구현한다는 것은 함수 분기의 입장에서 봤을 때 두 가지 의미를 가진다고 볼 수 있습니다.

1. 기능의 확장 - 어떤 기능 수행을 위한 구현 방법이 다양해지는 것, 즉, 분기 범위 확대. 압축 프로그램이라면 지원되는 압축 알고리즘의 수가 늘어나는 것이 이에 해당됩니다.
2. 기능(혹은 인터페이스)의 추가 - 기존의 기능 외에 새로운 기능이 추가되는 것(분기점 추가), 예를 들어 압축 하기, 압축 풀기 외에 압축파일 외부에 전송하기라는 기능이 추가되는 것은 기존 기능에서의 분기가 아니라 새로운 기능이 만들어 진다는 것을 의미하며 동시에 새로운 분기의 가능성이 생겼다는 것을 뜻합니다. 압축 파일 외부 전송하기 라는 기능은 메일 전송과 FTP 전송으로 분기될 수 있습니다.

따라서 우리는 어떤 기능을 구현 할 때 이것이 위 두 경우 중 어느 부분에 속하는 지 먼저 확인해야 합니다. 왜냐하면 어느 부분에 속하느냐에 따라 구현이나 설계 구조에 있어서 큰 차이를 보일 수 있기 때문입니다. 이전에 '상속, 합성 그리고 다중 상속'이라는 글에서 가급적 상속 보다 합성을 이용하라는 말을 했었습니다. 따라서 아마도 새로운 인터페이스를 추가할 때 우리는 이에 대한 새로운 클래스를 만들고 이를 합성하는 구조로 갈 것입니다. 그리고 해당 기능이 어떻게 확장될 것인가(분기 범위 확대)에 따라 해당 클래스의 내부 구현 구조를 결정해야 합니다. 예를 들어 이미 구현된 어떤 클래스 구조에 새로운 기능을 추가할 때 우리는 아래처럼 합성 구조를 이용하여 기능을 추가할 수 있습니다.



그리고 이렇게 되면 새로 추가된 FuncBase라는 기능이 확장될 때 마다 확대되는 가변성을 아래와 같은 상속구조를 이용하여 처리할 수 있습니다.



이러한 구조는 높은 유연성과 응집력, 낮은 결합도를 유지할 수 있기에 좋은 구조입니다. 때문에 디자인 패턴에서는 이런 구조를 선호하며 대부분 이런 식으로 객체 지향 설계가 이루어 집니다. 그러나 이런 구조는 **새로운 클래스 구조의 추가 및 객체 생성 문제에 따른 높은 초기 비용을 필요로 합니다.** 만약 매우 단순한 기능에 대한 확장을 이런 식으로 처리하려 한다면 비록 구조적으로는 안정적으로 보일지 모르지만 코딩상으로는 무척 비효율적인 방법이 될 것입니다. (게다가 클래스의 개수가 지나치게 많아져 오히려 유지 보수에 문제점이 발생할 수 있습니다.)

우리는 앞서 함수 분기를 위한 많은 방법들에 대해 이야기 했었습니다. 그리고 각각의 장/단점에 대해 알아 보았습니다. 만약 간단한 함수 분기가 필요하다면 switch문이나 if / else if 문을 사용해도 충분할 것입니다. 또한 각각의 함수 형태가 동일하고 하나의 함수에서 처리 가능하지만 매우 큰 규모의 분기 구조가 필요하다면 함수 테이블을 이용하는 것이 가장 좋을 것입니다. 물론 이런 함수 분기가 정적으로 이루어 진다면 정적 다형성이나 템플릿을 이용해야 합니다. 상속 구조는 이런 것들을 이용하는 것보다 확실히 이득을 볼 수 있는 경우에만 취하는 것이 좋습니다.

문제는 이런 판단을 얼마나 정확하게 할 수 있겠느냐는 것입니다. 소프트웨어는 항상 변화하며 여자의 마음보다도 더 알 수 없는 게 클라이언트의 마음입니다. 따라서 지금은 별다른 문제가 없어 보이는 부분이 프로젝트의 막바지에 이르러 갈 길 바쁜 우리의 발목을 잡을 지 모릅니다. 철저한 요구 사항 분석 및 사전 설계에 의해 이러한 문제를 어느 정도 해결할 수 있을 지 모르나 이 세상에 완벽한 설계는 없는 법입니다. 따라서 우리는 최대한 변화에 유연한 구조를 갖도록 노력해야 합니다.

결국 이런 상황에서 우리가 취할 수 있는 최선의 방법은 새로이 추가되는 인터페이스가 있고 그것이 기능 확장의 가능성을 가지고 있다면 별도의 클래스로 캡슐화해야 하며 이 때 내부적으로 어떤 식의 분기 구조를 가질 것인지는 외부에 절대 노출하지 않는 것입니다. 그리고 이런 구조를 통해서만이 앞으로 어찌 될 지 모르는 미래의 기능 확장 및 변경에 대비해 다양한 분기 구조로 변형할 수 있습니다. 이전 글에서도 언급했었지만 모든 상황에 완벽한 해결책은 없습니다. 다만 현재 상황에 더 나은 해결책만이 존재할 뿐입니다. 따라서 **시시 각각 변화하는 상황에 맞는 더 나은 해결책을 적용하기 위해 가장 유연하게 대처할 수 있는 분기 구조**를 만들도록 노력해야 합니다. 디자인 패턴에서 주장하는 인터페이스를 이용한 설계는 바로 이런 것을 말합니다.

<글쓴이 : 이은조(gimmesilver@hanmail.net, http://agbird.egloos.com)>