C++에서 메시지 프로토콜의 점진적인 확장 기법

김은주†

요 약

본 논문에서는 다형적 변수에 대한 메시지 전달의 유효성을 정적 형검사로 간주하는 C++ 에서 메시지 프로토콜을 점진적으로 확장할 수 있는 프로그래밍 기법을 제시하였다. 제시한 기법은 메시지의 유형 , 인수, 및 유효성을 동적으로 검사할 수 있는 프로시듀어인 메시지 핸들러라는 관점에서 메시지 프로토콜을 해석하도록 하였다. 이러한 해석의 결과 메시지 프로토콜의 점진적 확장에 효과적으로 대처할 수 있을 뿐 아니라 기존의 객체지향 프로그래밍의 모든 요소들을 충실히 모방할 수도 있었으며, 객체의 그룹에 어떤 메시지를 전달하는 기능과 같은 고수준의 프로시듀어 작성도 가능하게 되었다.

An Idiom for Incremental Extension of Message Protocol in C++

Eun-Ju Kim†

ABSTRACT

In this paper, I present a programming idiom for extending message protocols incrementally in C++, where effectiveness of message transmission is regarded as static binding. Our techniques analyze message protocols with a message handler that examines message types, parameters, and effectiveness dynamically. The result of this analysis is not only to cope with incremental extension of message protocols effectively but also to simulate all essential elements of the object-oriented programming. This result also makes it possible to write high level of procedure like message transmission to object groups.

1. 서 론[1)]

객체지향 프로그래밍은 기존 코드의 재사용과 비 파괴적인 확장을 용이하게 하는 효과적인 프로그래밍 패러다임이다. C 언어의 확장인 C++ 언어[6]에서도 클래스, 상속성, 동적 바인딩, 다형성 등의 기능을 제공하여 객체지향적 프로그래밍을 적극적으로 지원하고 있다.

C++ 언어에서 어떤 클래스의 모든 객체가 인식할 수 있는 메시지의 유형과 인수 및 결과치 등("메시지 프로토콜")은 클래스를 정의할 때 완전하게 기술하여야 한다. 그 결과 메시지 전달의 유효성을 정적으로 검사할 수 있으며, 메소드 바인딩의 효율을 향상시킬 수 있다는 장점을 제공하지만 하부 클래스에서 추가한 새로운 메시지를 상부 클래스의 다형적 변수를 통해 전달하는 것이 허용되지 않는 문제점도 있다. 이런 제약으로 인하여 다형적 변수에 크게 의존하면서도 모든 객체의 메시지 프로토콜을

1)정회원: 동명정보대학교 정보공학부 정보통신공학과 교수
논문접수: 2003년 5월24일, 심사완료: 2003년 6월6일

사전에 완전히 예측할 수 없는 프로그램을 개발하는 작업이 대단히 어렵게 된다. 프레임워크 라이브러리를 개발하고자할 때, 이 라이브러리를 다루는 일반적인 알고리즘은 존재하지만 다루는 대상이나 그 대상에 적용할 수 있는 메시지의 패턴을 미리 알 수 없는 경우, 다형적 변수를 통한 메시지 전달에 많은 제약이 따른다. 그리고 프로그램의 일부를 프로토타입의 형태로 개발하고 이를 점진적으로 확장하고자 하는 경우도 메시지 프로토콜의 동적 확인 기능이 필요하다.

본 논문에서는 메시지 프로토콜을 메시지 핸들러라는 관점에서 해석하여 이러한 문제점을 해결하고자 하였다. 메시지 핸들러는 전달된 메시지의 유형과 인수, 적합성 등을 동적으로 검사할 수 있는 프로시듀어이다. 각 클래스는 메시지 프로토콜을 클래스에 완전하게 정의하는 대신 메시지 핸들러만을 정의한다. 이와 같은 관점은 메시지 프로토콜을 컴파일러가 정적으로 완전하게 검사할 수 있는 내용으로 해석하는 대신 메시지를 핸들러의 인수로 전달할 수 있는 값으로 대응시켜 동적으로 확인하게 한 것이다.

본 논문에서는 메시지 핸들러를 사용한 관용적인 표현이 점진적으로 확장되는 메시지 프로토콜을 지원하는 환경에서 다형적 변수를 통하여 확장된 메시지를 전달하는 경우 프로그램 재사용을 위한 문제를 효과적으로 해결할 수 있음을 보였다. 또한 메시지 핸들러가 기존 객체지향 프로그래밍의 모든 기본 요소들을 충실히 모방할 수 있음도 보였다.

2. C++ 언어 재사용 기능의 문제점과 제약의 원인

C++ 언어는 C 언어에 데이터 추상화와 객체지향 프로그래밍 기능을 추가한 혼합언어이다. C++는 C언어에 클래스, 상속성, 동적 바인딩과 다형성등의 유용한 기능을 제공하여 프로그램의 재사용과 확장을 용이하게 한다.

2.1 C++ 언어 재사용 기능의 문제점

어떤 클래스의 "메시지 프로토콜"은 그 클래스의 모든 인스턴스들이 인식하는 메시지들의 공용 인터페이스를 의미한다. C++에서는 메시지 프로토콜을 클래스의 정의에 명시하는 것이 기본적인 프로그래밍 기법이다. 그래픽 시스템을 위한 프로그램에서 모든 그래픽 객체를 위한 추상 클래스 Figure는 다음과 같다.

```
class Figure {
public:
    virtual void  draw(void) = 0;
    virtual void  move(int, int) = 0;   };
```

클래스 Figure에서 파생된 모든 유형의 객체들은 draw와 move 메시지를 처리할 수 있어야 함을 클래스의 정의에서 직접 확인할 수 있다. 멤버함수만을 사용하는 표준프로그래밍 기법에서는 Figure와 같은 다형적 변수를 위한 베이스 클래스는 하부 클래스들의 공통적인 프로토콜을 사용 이전에 완전히 기술하여야 하며, 베이스 클래스의 메시지 프로토콜을 변경하거나 확장할 필요가 있는 경우 전체 프로그램을 다시 컴파일하여야 한다. 주어진 예에서 Figure 클래스에서 파생된 Rectangle 클래스의 경우, 이 유형의 객체는 그 내부를 임의의 색상으로 채울 수 있도록 하는 메시지 fill을 추가로 인식하도록 할 경우 다음과 같은 다형적 변수의 사용은 컴파일러에 의해 에러로 간주된다.

```
Figure & fig = Rectangle(...);
fig.fill(COLOR_RED);  // ERROR!!!
```

그 결과 Figure 클래스에 대한 수정을 필연적으로 요구하게 된다. 원시 프로그램이 제공되지 않거나, 프레임워크 라이브러리 개발과 같이 모든 객체의 메시지 프로토콜

을 완전하게 예측할 수 없는 문제인 경우에는 이러한 수정이 가능하지도 않다.

2.2 C++ 언어 재사용 기능의 제약의 원인

지금까지 소개한 문제점의 원인은 C++에서는 메시지 프로토콜을 정적으로 완전하게 검사할 수 있는 기능으로 설계하였기 때문이다. 메시지 프로토콜의 선언 수단인 클래스는 구조체의 확장으로 설명될 수 있으며, 적용 가능한 메시지의 종류는 구조체의 필드명칭에, 메시지의 문법은 각 필드의 형에 대응하게 된다. 그러므로 새로운 메시지를 인식하게 하기 위해서는 구조체에 새로운 필드를 선언하는 것과 같은 프로그램의 수정이 필요한 것은 당연한 결과이다. 이와 같은 설계는 메시지 전달의 유효성을 정적으로 검사하고 효율을 개선하기 위한 목적에서 보면 효과적일 수도 있으나 프로그램의 확장과 재사용 관점에서는 심각한 제약이기도 하다.

메시지의 타당성 여부를 전적으로 실행시에 검사하는 Smalltalk[2]와 같은 언어에서는 이러한 제약이 전혀 없어서 점진적인 프로그램 확장이 대단히 용이한 대신 메시지 전달의 유효성을 실행 이전에 전혀 확인할 수 없는 문제가 있다.

2.3 관련 연구

C++ 언어에서 활용될 수 있는 다수의 관용적인 표현을 소개한 Coplien의 연구[1]에서도 점진적인 프로토콜 확장 기법이 소개되고 있다. 이 연구에서는 클래스에 기반을 둔 C++언어의 근본적인 사고와는 달리 프로토타입에 기반을 둔 exemplar라는 새로운 사고를 요구하며, 메시지 프로토콜뿐만 아니라 객체의 유형과 동작까지 변경될 수 있는 극도의 유연성을 고려한 기법을 제시하였다. 그 결과 프로그래밍 구성이 전형적인 C++ 프로그램과는 많이 다르며, 비효율적이고 안전하지도 않다.

한편 wirth와 Gutknecht의 Project Oberon[9]은 객체지향 언어 Oberon 언어[5]에서 본 연구에서와 유사한 기법을 이용하여 그래픽 에디터를 개발하는 예제를 소개하였다.

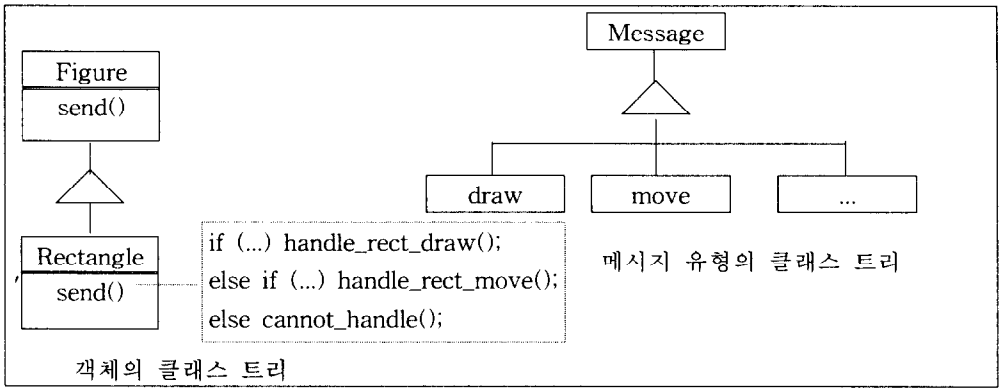
3. 확장 가능한 메시지 프로토콜

여기에서는 C++에서의 정적 형검사 및 컴파일 기능에 의한 안전성과 효율성을 유지하면서도 Smalltalk에서와 같은 유연성을 얻기 위한 프로그래밍 기법을 소개하기로 한다.

3.1 메시지 핸들러

C++에서 메시지 프로토콜의 선언 수단인 클래스는 구조체에, 적용 가능한 메시지의 종류는 구조체의 필드 명칭에 대응함을 설명한 바 있다. 이와 달리 본 논문에서는 메시지 프로토콜을 "메시지 핸들러"라는 관점에서 해석하고자 한다. 메시지 핸들러는 적용 가능한 메시지의 종류를 동적으로 판별할 수 있는 프로시듀어를 의미한다. 메시지 프로토콜을 클래스의 정의에 완전하게 기술하는 대신 메시지 핸들러 만을 클래스에 정의하도록 한다. 다음 프로그램에서 send 멤버함수가 Figure 클래스를 위한 메시지 핸들러이다. is_kind_of는 메시지 유형을 판별하기 위한 연산자로 가정한다.

```
class Figure { virtual void send(message); };
void Figure::send(message msg) {
   if (msg is_kind_of draw)) handle_draw();
   else if (msg is_kind_of move)
        handle_move();
   else cannot_handle();      }
```



(그림 1) 객체와 메시지 유형의 트리

이는 메시지의 종류를 구조체의 필드와 같이 컴파일러가 정적으로 확인할 수 있는 내용으로 보는 대신 프로시듀어의 인수로 전달할 수 있는 값으로 대응시켜 동적으로 확인하도록 관점을 변경한 것이다. 직사각형을 위한 클래스 Rectangle을 Figure에서 파생시키는 경우는 다음과 같다.

```
class Rectangle : public Figure { ... virtual
void send(message); ...};
void Rectangle::send(message msg) {
    if (msg is_kind_of draw)
      handle_rect_draw();
    else if (msg is_kind_of move)
          handle_rect_move();
    else cannot_handle(); }
```

이상을 요약하면 메시지 핸들러를 위한 관용적인 표현은 다음과 같다. 각 클래스는 send라는 멤버만 구현하도록 한다. 이 멤버는 적용할 메시지를 인수로 하여 이를 동적으로 판별, 적절한 기능을 수행하도록 한다. 메시지 프로토콜은 클래스의 작성자가 아니라 send 멤버 작성 시에 결정된다. 반면 다형적 변수와 동적 바인딩을 활용하면 적절한 핸들러를 선택하게 할 수 있다.

메시지 핸들러를 구현하기 위해서는 전달 가능한 메시지의 표현, 선달된 메시지의 판별, 및 메시지 인수의 추출을 위한 기능이 각각 필요하다. 메시지는 프로그램에서 허용되는 메시지인지의 여부를 컴파일러가 검사할 수 있어야 한다. 또한 전달된 메시지의 유형을 객체가 동적으로 확인할 수 있어야 하며, 인수를 표현할 수 있어야 한다. 이러한 목적을 모두 만족하는 표현 방법은 메시지를 객체의 일종으로 간주하는 것이다.

전달 가능한 모든 메시지는 message 클래스에서 파생된 클래스의 객체여야만 한다. 이 클래스는 메시지의 내용을 전혀 포함하고 있지 않으며 다음과 같이 정의된다.

```
class message {};
```

예를 들어 draw라는 메시지 유형을 프로그램에 추가하고자 할 경우에는 다음과 같이 정의할 수 있다.

```
class draw: public message { };
```

여기에서 특기할 사항은 draw라는 메시지가 특정 객체 유형에 종속되지 않고, 이 메시지를 처리하고자 하는 모든 유형의 객체나 클래스에 의해 공유된다는 점이다. 이들을 message 클래스에서 파생하도록 함으로써 메시지의 허용가능성을 컴파일러에게 인식시킬 수 있도록 하였다는 점에서도 중요하다. 만일 정수나 문자열로 메시지를 표현하였다면 그 값을 조회하기 전에는 메시지의 허용여부를 알 수 없으므로 컴파일러에 의한 검사가 불가능하게 될 것이다. 인수를 가진 메시지의 표현은 생성자를 가진 메시지 유형으로 정의하면 된다. 그래픽 객체를 이동시키기 위한 메시지의 클래스

move는 다음과 같이 정의될 수 있다.

```
class move : public message {
public:
   move(int, int); };
```

지금까지 소개한 객체의 클래스와 메시지 클래스의 아이디어를 객체 모델 기호법(Object Model Notation)[4]으로 표현하면 (그림 1)과 같다. (그림 1)에서 메시지의 유형이 Figure나 Rectangle에 직접 연관되지 않고 메시지 핸들러의 인수로 전달 할 수 있는 방법을 통해서 간접적으로 연관됨을 알 수 있다.

멤버함수만을 사용하는 기존의 프로그래밍 기법에서는 객체수준의 메시지 허용여부가 컴파일러에 의해 검사되었으나 본 연구에서의 기법은 핸들러에 의해 검사된다. 메시지 허용여부의 검사는 메시지의 판별기능에 의해서 수행된다. 메시지를 객체로 표현하였으므로 메시지의 판별은 동적 형 확인 기능(RTTI)[7]을 사용할 수 있다. 그리고 동적 형 확인 기능이 지원되지 않는 시스템에서는 "switch-on-type"[8]을 이용하면 된다.

핸들러에 의해 전달되는 메시지를 포인터형으로 제한할 경우 다음과 같이 dynamic_cast 연산자를 사용하여 메시지 판별기능을 구현할 수 있다.

```
void Figure::send(message * msg) {
      if (dynamic_cast<draw*>(msg))
            handle_draw();
      else if (dynamic_cast<move*>(msg))
            handle_move();
      else ....   }
```

이런 방법으로 구현된 핸들러는 다음 예에서와 같이 메시지를 전달할 때마다 동적 할당이 필요하게 된다.

```
Figure *fig = new ...;
fig->send(new draw());
fig->send(new move(100,50));
```

동적으로 할당된 객체의 효과적인 처리가 보장되지 않는 한 이와 같은 방법은 기억장소의 누수현상을 막기가 대단히 힘든 문제점이 있다. 이러한 문제점은 dynamic_cast 대신 typeid를 사용하면 해결될 수 있다.

```
void Figure::send(message& msg) {
      if (typeid(msg) == typeid(draw))
         handle_draw();
      else if (typeid(msg) == typeid(move))
            handle_move();
      else ... }
Figure& fig = ...;
fig.send(draw());
fig.send(move(100,50));
```

그러나 dynamic_cast에 비해 typeid는 메시지 유형을 판별하는 기능만 수행하므로 메시지 인수 추출을 위한 코드 작성을 어렵게 하는 문제가 있다. 이는 약간의 추가적인 프로그래밍 노력으로 해결될 수 있다. 메시지의 인수를 추출하기 위해서는 해당하는 메시지의 유형으로 형 변환할 필요가 있다. 메시지 유형의 검사는 typeid에 의해 이미 수행되었으므로 안전하면서도 수행부담이 적은 static_cast를 사용할 수 있다.

```
void Figure::send(message& msg) {
  ...
  if (typeid(msg) == typeid(move)) {
    move& mm = static_cast<move&>(msg);
    // use mm instead of msg from now on
  } ... }
```

무분별한 RTTI의 사용은 객체지향 프로그래밍의 원칙을 위배하는 것으로 간주된다. 그러나 본 논문에서의 RTTI 사용은 메시지에 대한 메소드의 동적 바인딩 원칙을 전혀 위배하지 않는 새로운 프로그램밍의 관례일 뿐이다.

3.2 메시지 핸들러와 객체지향 프로그래밍

메시지 핸들러는 기존의 프로그래밍 기법에 비해 메시지 프로토콜과 메소드를 상속하거나 재 정의할 때 더 많은 유연성을 부여한다. 상부 클래스의 모든 메시지 프로토콜과 메소드를 수정 없이 상속하기 위해서는 하부 클래스에서 send멤버 함수를 정의

하지 않으면 된다. 반면 일부 메시지에 대해서는 새로운 메소드를 사용하고 나머지 메시지들에 대해서는 상부 클래스의 객체와 동일하게 대처하기 위해서는 다음과 같은 기법을 사용할 수 있다.

```
void Rectangle::send(message& msg) {
   if (typeid(msg) == typeid(draw))
      handle_rect_draw();
   else Figure::send(msg);
      //기타 모든 메시지 상속
}
```

동일한 메시지에 대한 메소드를 하부 클래스에서 재정의 하는 것은 대단히 용이하다. 상기한 예에서 Rectangle 형의 객체에 draw 메시지가 전달되면 Figure에서의 메소드 대신 handle_rect_draw 라는 멤버를 사용하여 대응하게 한 것이 그 예이다. 만일 handle_rect_draw의 처리과정이 Figure의 draw 기능을 필요로 하는 경우에는 다음과 같은 방법으로 대처할 수 있다.

```
void Rectangle::send(message& msg) {
    if (typeid(msg) == typeid(draw)) {.....
       Figure::send(msg);   // up call
    ..... }
    .... }
```

상부 클래스의 메소드가 하부 클래스의 메소드를 호출하는 경우를 down call이라고 한다. 모든 Figure 객체에 erase 메시지가 전달된 경우 (draw를 위한 색상 지정이 가능하다고 가정할 경우) 배경 색상으로 draw 하는 메소드를 사용할 수 있다.

```
class erase: public message {};
void Figure::send(message& msg) {  .....
  else if (typeid(msg) == typeid(erase)) {
         send(draw()); }
  .... }
```

Rectangle 클래스의 핸들러에서 erase 메소드를 재 정의하지 않는 경우 이 메시지가 올바르게 동작하기 위해서는 send(draw())에서 draw 메시지를 Rectangle 클래스에서 정의된 핸들러가 처리하도록 하여야 한다. 즉 fig에 Rectangle 유형의 객체가 기억되어 있을 경우 fig.send(erase())는 Figure::send(erase())를 호출하며, 이는 다시 Rectangle:: send(draw())를 호출하여야 한다. 여기에서 send(draw())는 this->send(draw())와 같은 의미이므로 send가 객체에 의해 동적으로 검색되도록 하면 된다. 그러므로 down call은 메시지 핸들러를 가상함수로 선언함으로써 효과적으로 해결할 수 있다.

다음 프로그램에서는 Rectangle과 Circle 클래스에서 공히 파생된 RCHybrid클래스에 대한 예를 통해서 메시지 핸들러와 다중 상속의 관계를 설명할 수 있다. RCHybrid 객체는 draw 메시지에 대해서는 Rectangle과 Circle의 draw 메시지를 공히 적용하며, fill 메시지에 대해서는 Circle에 대해서만 적용하기로 하고, 기타 모든 메시지는 Rectangle 혹은 Circle 가운데 주어진 메시지를 인식하는 핸들러가 처리하도록 한다. 이 경우 모든 상부 클래스의 핸들러 가운데 msg에 대응하는 가장 적절한 핸들러를 검색하고 이를 호출하는 기능을 수행하여야 한다. 이 문제를 해결하기 위해서는 다음과 같이 메시지 핸들러로 하여금 전달된 메시지의 처리 여부를 결과치로 반환하게 함으로써 해결할 수 있다.

```
class RCHbrid: public Rectangle, public
Circle { virtual int send(message&); };
int RCHbrid::send(message& msg) {
   if (typeid(msg) == typeid(draw)) {
      Rectangle::send(msg);
      Circle::send(msg);
      return(1); }
   else if (typeid(msg) == typeid(fill)) {
            Circle::send(msg);
            return (1); }
   else
         return (Rectangle::send(msg) ||
                  Circle::send(msg)); }
```

메시지의 처리는 메시지 전달의 체인을 따라 최상위 클래스까지 파생될 수 있다. 예를 들어 Figure 객체들이 전혀 처리할 수

없는 메시지가 RCHybrid형의 객체에 전달된 경우 Rectangle과 Circle을 통해 Figure에까지 전달 될 수 있다. 이 경우에 대비하기 위하여 Figure 클래스를 위한 메시지 핸들러는 다음과 같은 구조를 취할 수 있다.

```
int Figure::send(message& msg) { ....
  else  return (0); } //  cannot handle msg!
```

3.3 메시지 프로토콜의 점진적 확장

일반적인 Figure 객체에는 적용할 수 없으나 Rectangle 객체에는 적용할 수 있는 새로운 메시지의 정의는 핸들러로 하여금 그 메시지에 대한 경우를 추가하기만 하면 된다. 새로운 메시지 유형이 미리 정의되지 않은 경우에는 메시지의 정의를 동반할 수도 있다.

```
class fill: public message{ public: fill(color);};
void Rectangle::send(message& msg) {....
  else if (typeid(msg) == typeid(fill))
        handle_rect_fill(); ...   }
```

메시지 핸들러는 다음 예에서와 같이 하부 클래스에서 추가로 확장한 메시지를 다형적 변수를 통하여 호출하는 문제를 위한 효과적인 해결책을 제공한다.

```
Figure& fig = Rectangle(...);
fig.send(fill(COLOR_RED));  // it works!
```

일반적으로 하부 클래스에서 새로운 메시지를 처리하도록 확장할 경우에는 이를 상부클래스의 공용 인터페이스에 반영하여야 할 경우가 있음을 앞서 지적하였다. 반면 메시지 핸들러를 사용하면 메시지 프로토콜이 확장될 경우 메시지 트리와 메시지 핸들러의 구현이 확장될 뿐이며, 객체 클래스의 인터페이스는 전혀 변경될 필요가 없다.

4. 제안한 기법의 평가

여기에서는 본 연구에서 제안한 기법의 일반성과 문제점을 소개하고 다른 언어에서의 메시지 프로토콜 확장 기법과를 비교한다.

4.1 메시지 핸들러의 일반성

메시지 유형을 정적인 요소로 간주하는 기존의 멤버함수 기법에서는 "모든 객체에 대하여 주어진 메시지를 적용하라"는 의미의 프로시듀어를 작성할 수 없으며 fill_all, draw_all 등과 같은 별도의 함수로 작성하는 수밖에 없다. 뿐만 아니라 fill과 draw의 인식 가능성이 Figure 클래스에 미리 선언되지 않은 경우에는 이러한 프로시듀어의 작성조차 금지된다. 그러나 메시지 핸들러를 이용하면 이러한 프로시듀어를 작성하여 라이브러리에 포함시킬 수도 있다.

```
void sens_all( list<Figure*>fig_lst, message&
msg) {
  Figure *fig;
  fig_lst.rewind();
  while (fig_lst.get_next(fig))
        fig->send(msg);
}
```

여기서 rewind, get_next는 iterator함수[3][10]를 사용하였으며, 여기서 특기할 사항은 전달되는 메시지 유형이 send_all 프로시듀어의 인수로 사용되었다는 점이다. 즉 메시지를 컴파일러에 의해 완전히 검사되는 형 정보로 간주하는 대신 데이터의 일종으로 감주함으로써 이와 같은 유연성을 얻은 것이다. 예를 들어 fig_lst에 포함된 모든 객체들에 대하여 fill과 draw 메시지를 차례로 전달하는 기능을 프로그래밍 한 예는 다음과 같다.

```
// list<Figure*> fig_lst;
send_all(fig_lst, fill(COLOR_RED));
send_all(fig_lst, draw());
```

상부 클래스에서 인식하는 메시지에 대하여 하부 클래스에서 모두 대응하게 하는 것은 인터페이스 상속의 원리에서 볼 때 바람직한 것이다. 그러나 어떤 필요에 의해서 특정한 메시지에 대해서는 대응하기를 원하지 않는 경우 메시지 핸들러는 효과적인 해

<표 1> 멤버함수, 메시지 핸들러와 Smalltalk의 비교

	메시지 유형 판별시기	메시지의 가시성	메시지-메소드 바인딩시기	메시지 유효성 검사시기	인수의 적합성 검사시기	메소드 호출비용
C++ 멤버함수	정적	클래스와 파생클래스	정적+동적	정적	정적	일정
C++ 메시지 핸들러	동적	프로그램 전역	동적	동적+정적	정적	상속성 깊이에 비례
Smalltalk	동적	프로그램 전역	동적	동적	동적	상속성 깊이에 비례

결책을 제공한다. 예를 들어 Rectangle 클래스의 객체로 하여금 draw와 fill 메시지에 대해서만 대응하게 하도록 핸들러를 작성한 예는 다음과 같다.

```
void Rectangle::send(message& msg) {
  if (typeid(msg) == typeid(draw)) ...
  else if (typeid(msg) == typeid(fill))....
  // but no else clause
}
```

메시지 핸들러는 어떤 객체가 전달된 메시지를 처리하였는지의 여부를 동적으로 확인 할 수 있는 장점도 제공한다.

```
if (!fig_send(fill(COLOR_RED)))
   try_other_method();
```

예에서는 다형적 변수 fig에 기억된 객체가 fill 메시지를 인식하지 못한 경우에는 try_other_method라는 프로시듀어를 호출하는 방법으로 대처하라는 의미가 된다.

메시지 핸들러는 C++에서 재사용이나 확장성을 저해하는 중요한 원인 하나를 원천적으로 제거하는 효과도 있다. C++에서는 메시지_메소드 바인딩이 정적으로 이루어지는 것이 묵시적인 약속이며, 가상함수로 선언된 경우에만 동적 바인딩이 발생한다. 이에 따라 베이스 클래스에서 정적 바인딩이 발생하도록 선언된 멤버함수를 하부 클래스에서 재정의 할 경우 베이스 클래스의 수정이 필요하다. 메시지 핸들러는 멤버 send 만을 가상함수로 선언함으로써 send가 처리 할 수 있는 모든 메시지에 대응하는 기능이 동적으로 선정되므로 메시지 프로토콜의 모든 멤버들이 가상함수로 선언된 것과 같은 효과를 얻을 수 있다.

4.2 다른 기법이나 언어와의 비교

Smalltalk은 동적 형검사를 채택한 언어로서 메시지 전달의 안전성을 실행 이전에 전혀 검사하지 않는 문제점이 있다. 반면 본 논문에서는 메시지 전달 가능성을 특정 클래스(및 그룹)에 국한하지 않고 프로그램의 모든 영역으로 확장한 점을 제외하면 정적 형 검사의 장점을 충분히 활용할 수 있다. 예를 들어 제 3장에서 소개한 기법에서 message 클래스에서 파생되지 않은 클래스의 객체가 메시지로 전달되는 경우를 컴파일러가 에러로 지적할 수 있다. 뿐만 아니라 메시지를 위한 인수의 개수와 형은 메시지 클래스의 생성자에 명시된 규약을 준수하여야 한다. 이와 같은 특징이 Smalltalk에서는 전혀 제공되지 않는다. 또한 C++에서는 send 함수를 컴파일 할 수 있고 메시지의 유효성을 정적으로 검사할 수도 있으므로 메시지 전달의 모든 과정을 전적으로 실행시에 처리하는 Smalltalk에 비해 실행 효율을 현저히 개선시킬 수 있다. <표 1>은 C++에서 기존의 C++ 멤버함수를 사용하는 경우와 본 논문에서 제시한 메시지 핸들러를 사용하는 경우와, Smalltalk에서 메시지를 처리하는 기법을 각각 비교한 것이다. 본 논문에서 제시한 기법은 C++기존 방법의 효율성과 안전성, 그리고 Smalltalk가 가지는 유연성을 절충한 효과를 얻을 수 있음을 알 수 있다.

본 논문에서 제안한 기법은 Project Oberon에서 사용된 기법과 본질적으로 유사하며, C++에서도 Oberon에서와 유사한 기법이 효과적으로 적용될 수 있음을 보인 것이다. 그러나 메시지 핸들러와 객체지향 프로그래밍의 기본 기능과의 연관성을 심도 있게 조사한 것, 다중 상속과 메시지 핸들러를 연관 시킨 것, 객체지향 프로그래밍의 원칙상 일반적으로 사용이 금기시되는 RTTI 기능이 활용될 수 있는 분야를 보인 것 등은 본 연구의 고유한 결과이다.

본 연구에서 제시한 기법은 메시지의 판별 과정에 임의적인 순서가 존재한다. 그러므로 if문의 체인에서 먼저 나오는 메시지 유형에 비해 나중에 나오는 메시지 유형은 비효율적일 수 밖에 없다. 메시지 프로토콜을 구조체에 대응시킨 기존의 C++ 전략에서는 가상함수의 호출을 위한 부담이 상속성 계층의 깊이에 관계없이 일정하였으나 메시지 핸들러는 깊이에 비례하는 검색 노력을 요구하는 문제점이 있다. 선별적인 메시지의 처리를 위한 핸들러 작성의 경우 의도하지 않게 메시지 판별 문장을 실수로 누락하더라도 이를 정적으로 검사할 수 없다. 또한 모든 메소드가 공용 인터페이스로 간주되므로 private나 protected 가시성을 가진 메소드의 작성이 어려운 문제점도 있다.

5. 결론

본 논문에서는 C++와 같이 다형적 변수에 대한 메시지 전달의 유효성을 정적 형검사로 간주하는 언어에서 메시지 프로토콜을 점진적으로 확장할 수 있는 관용적 표현을 제시하였다. 제시한 관용적인 표현은 메시지의 유형, 인수, 및 유효성을 동적으로 검사할 수 있는 프로시듀어인 메시지 핸들러라는 관점에서 메시지 프로토콜을 해석하도록 하였다.

메시지 핸들러는 메시지 프로토콜의 점진적 확장에 효과적으로 대처할 수 있을 뿐만 아니라 기존의 객체지향 프로그래밍의 모든 요소들을 충실히 모방할 수도 있게 되었으며, 객체의 그룹에 어떤 메시지를 전달하는 기능과 같은 고수준의 프로시듀어 작성도 가능하게 되었다. 이러한 요소들은 프로그램에서 사용될 객체의 유형과 메시지 프로토콜을 사전에 완전하게 예측할 수 없는 프레임 워크 라이브러리의 개발 등에 필수적으로 요구되는 기능이다. 제시한 기법은 동적 형 확인 기능에만 의존하고 있으므로 이 기능을 제공하는 어떤 객체지향 언어에도 적용될 수 있다.

본 연구에서 제안한 기법은 메시지 전달의 유효성이 클래스에 국한되지 않고 프로그램 전역으로 확대되며, 메시지 핸들러에 의해 동적으로 검사된다. 이는 본 연구에서 추구하는 유연성을 제공하는 주된 원인이기는 하지만 프로그래머의 실수에 의해 특정 메시지에 대한 대처 기능이 누락된 경우의 검사가 정적으로 이루어지지 않는 문제점도 있다.

참고문헌

[1] J. O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley Publishing Company, 1992.

[2] A. Goldgerg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison- Wesley Publishing Company, 1983.

[3] Kim, M.H., "A New Itearion Mechanism for The C++ Programming Language," ACM SIGPLAN Notices, Vol.30, No.2, pp.17-24, 1991.

[4] J. Rambaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorenson, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[5] M. Reiser and N. Wirth, *Programming in Oberon*, Addison-Wesley Publishing Company, 1992.

[6] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Company, 1991.

[7] B. Stroustrup and D. LenKov, "Run-Time Type Identification for C++," Proc. USENIX C++ Conference, Portland, OR., Aug., 1992.

[8] G. V. Vaughan, B. Elliston, T. Tromey and I. L. Taylor, *The Goat Book,* New Riders Publishing, 2000.

[9] N. Wirth and J. Gutknecht, *Project Oberon*, Addison-Wesley Publising Company, 1992.

[10] 김은주, 김 명호, "일등급 C++ 열거자를 이용한 프로그래밍," 정보과학회논문지 (B), 제23권 6호, pp.647-660, 한국정보과학회, 1996.

김 은 주

1984 경북대학교 전자공학과 (전산공학, 공학사)
1986 경북대학교 전자공학과 (전산공학, 공학석사)
2003 경북대학교 컴퓨터공학과(공학박사)
2000~현재 동명정보대학교 정보공학부 정보통신공학과 전임강사
관심분야: 멀티패러다임 프로그래밍, 병렬 알고리즘, 병렬 제어추상화
E-Mail: ejkim@tit.ac.kr