클래스 계층구조 슬라이싱을 이용한 C++ 프로그램 최적화에 관한 연구

김 운 용[†] · 정 계 동[††] · 최 영 근[†††]

요 약

본 논문에서는 C++ 클래스 계층구조(상속관계를 가진 클래스들의 모음)를 대상으로 객체 지향 언어의 특성인 단일/다중 상속, 정적/동적 바인딩, 함수중복/함수재정의(Overloading & Overriding), 순수가상/가상함수, 생성자 문제를 고려하여 멤버 데이터와 멤버함수를 최적화 할 수 있는 알고리즘을 제안한다. 프로그램 계층 구조와 그 계층 구조를 사용하는 프로그램은 일반적으로 클래스 계층 구조의 부분적인 기능만을 사용하기 때문에 많은 구성요소를 포함하는 클래스들에서 불필요한 기능을 제거하는 것이 필요하게 되었다. 지금까지 연구되어 왔던 고전적인 슬라이싱이나 다른 변형된 슬라이싱은 출력데이터를 선택하고 그와 관련된 프로그램 문장을 포함하는데 초점을 맞추고있다. 그 대상은 대부분 구조적 프로그램 언어로 이루어졌으며 이러한 슬라이싱은 주로 오류 검출, 소프트웨어의 유지보수, 유연한 테스팅을 위한 주제로 연구가 되어 왔다. 본 논문에서는 그 대상 범위를 객체 지향 언어로 확장시키고, 분석단계에서 테이블 구성형태를 링크형태로 구성함으로써 보다 정보 관리의 효율을 높일 수 있고, 이 테이블을 이용한 최적화 시스템 구현을 통해 필요한 알고리즘을 제시하였다. 이러한 과정을 통해 불필요한 멤버데이터, 멤버함수, 클래스 상속관계를 제거함으로 프로그램 코드의 간소화, 시스템 성능의 향상을 가져올 수 있다.

A Study on the Optimization of C++ Program Using the Class Hierarchies Slicing

Woon-Yong Kim[†] · Kye-Dong Jung[††] · Young-Keun Choi[†††]

ABSTRACT

This paper proposes an algorithm for class hierarchies which can optimize member data and member function. This algorithm considers single/multiple inheritance, static/dynamic binding, overloading/overriding, pure virtual/virtual function, and constructor on the hierarchy of C++ class. We need to eliminate unused function that possesses many component element, because the program uses a limited of function in class hierarchies. Previous works on slicing mainly focused on selecting output data and including the related program statement. It was consists of structured programming language and also centralized on error detection, maintenance, and flexible testing. In this paper, we extend to the object-oriented language, makes a linked-table for objects to raise the efficiency of information management , and proposes necessary algorithm for optimizing system. Through this process, we can obtain the simplification of program code and the progress of system performance by eliminating unused member data and member function.

† 준 회 원 : 광운대학교 대학원 전자계산학과
†† 정 회 원 : 광운대학교 대학원 전자계산학과
††† 정 회 원 : 광운대학교 전자계산학과 교수
논문접수 : 1998년 10월 14일, 심사완료 : 1999년 5월 3일

1. 서 론

객체 지향 소프트웨어 개발은 독립적이고 확장성을 가진 소프트웨어 부품들간의 관련성을 통하여 새로운 소프트웨어 개발을 위한 방안으로 제기되고 있다. 또한, 소프트웨어에 대한 일반적인 사용자의 요구를 충족시키기 위하여 많은 기능을 가지고 방대한 소프트웨어가 개발되고 있다. 그러나, 대체적으로 사용자는 소프트웨어의 일반적인 기능만 사용하는 경우가 많다. 많은 기능을 가지고 있는 소프트웨어에서의 필요한 기능만 남기고 불필요한 기능을 제거하여 구축할 경우, 다음과 같은 장점을 가질 수 있다.

첫번째로는 프로그램 전체 크기를 줄임으로써 실행 효율을 증대시킨다.

두번째로는 불필요한 기능을 제거함으로써 프로그램이 명확히 되고 디버그나 분석에 용이하다.

세번째로는 유사 시스템에 재사용될 프로그램을 최적화할 수 있으므로 소프트웨어 품질을 높일 수 있다.

네번째로는 최적화된 프로그램을 이용함으로 프로그램의 로딩시간 및 컴파일 시간이 감소한다.

이와 같은 장점은 프로그램의 대형화로 인한 현대에 중요한 이슈로 등장하였다. 본 논문에서는 객체 지향 언어인 C++ 언어를 대상으로 불필요한 클래스 정보의 멤버함수 및 멤버데이터를 제거하는 내용을 중심으로 고찰하고자한다. 본 논문의 구성은 다음과 같다. 제2절에는 관련 연구를 살펴보고, 제3절에서는 슬라이싱을 위한 프로그램 최적화 알고리즘을 제시하고, 제4절에서는 시스템적용의 예 및 비교, 분석을 보이며, 끝으로 제5절에서는 결론과 향후 연구 방향을 제시한다.

2. 관련연구

기존의 프로그램 슬라이싱의 대부분은 프로그램 유지보수에 이용되지만 이 특징을 이용한 재사용 측면에서는 프로그램의 그 기능이 전부 필요한 것이 아니기 때문에 실행되는 부분을 최적화하면 프로그램 실행 효율이나 가독성의 향상이라는 점에서 매우 유용하다.

2.1 고전적인 슬라이싱 기법

프로그램 슬라이싱의 기법은 변수의 의존 관계와 제어의 흐름을 분석하여 어떤 주어진 출력 변수들에 직접, 간접으로 영향을 미치는 프로그램 문장들을 추출해내는 기법으로, 이 방법[6][2]은 1984년 Weiser[16]에 의해 처음으로 소개되었다. 즉, 슬라이싱 기준 <n,V>가 주어졌을 때 "프로그램 문장 n이 수행되기 전에 변수 V에 영향을 미치는 문장은 무엇인가"에 대한 답을 제공하는 것이다. 이러한 고전적 문장슬라이싱의 개념을 이해하기 위해 (그림 1)과 같은 예제를 통해 설명한다. (그림 1)의 (A)는 sum과 factorial을 계산하는 프로그램이다. 그리고 (그림 1)의 (B)는 이 프로그램의 마지막 라인에 있는 factorial의 값과 관련된 슬라이싱을 보여준다. 이 예제에서 sum과 관련된 문장이 제거된다. 이렇게 함으로써 factorial과 관련된 에러에 관심을 가지는 프로그래머는 에러를 추적할 때 생략된 문장을 간주하지 않고 단지 factorial과 관련된 문장만을 디버그함으로써 디버깅의 효율성을 증대시킨다.

```
#include <iostream.h>
void main(){
    int n;
    cin >> n;
    int sum = 0;
    int factorial = 1;
    for(int i=1;i<=n;i++)
    {
            sum +=i;
            factorial *=i;
    }
    cout << sum << endl;
    cout << factorial << endl;
}
```

(A)

```
#include<iostream.h>
void main(){
    int n;
    cin >> n;
    //-------
    int factorial =1;
    for(int i=1;i<=n;i++)
    {
            //-----------
            factorial *=i;
    }
    //-------
    cout << factorial << endl;
}
```

(B)

(그림 1) 고전적 문장 슬라이싱

2.2 슬라이싱 연구의 방향

슬라이싱의 연구 방향의 목표는 정확한 슬라이싱 정보를 신속하게 생성하기 위한 방법을 개발하는 것이다. 슬라이싱을 위해 두 가지의 서로 다른 방법이 시도되었다. 첫째는 제어흐름 그래프(Control Flow Graph)를 사용하는 것으로 원시코드의 문장을 노드(node)로, 제어의 흐름을 에지(edge)로 하여 제어흐름 그래프를 생성해 낸 후 이를 이용하여 기준으로 주어진 문장n에 해당하는 노드로부터 후진(backward)으로 그래프의 에지를 따라가며 변수 V에 영향을 미치는 노드들을 찾아내는 것이다. 이는 초기에 많이 사용된 알고리즘으

로 간단하지만 생성 시마다 데이터와 제어의 흐름을 계산하며 그래프를 분석해야 한다는 비효율성을 단점으로 가진다. 둘째는 보다 효율적인 알고리즘으로 의존 그래프(Dependence Graph)를 사용하는 것이다. 원시코드의 문장은 노드로 데이터/제어 의존성이 에지로 표현되며 이 그래프는 원시코드로부터 만들어질 때 많은 시간을 요하지만 일단 한번 그래프를 생성하여 저장해 놓으면 매번의 슬라이싱 정보 생성은 선형 시간으로 충분하다는 장점을 가져 최근 많이 사용된다. 정확한 슬라이싱 정보 생성을 위한 알고리즘에 관해서도 많은 연구가 이루어졌고 아직도 추가적인 연구를 필요로 하는데, 여러 개의 프로시저가 관련된 경우, Goto 문이 존재하는 경우, 배열 및 포인터가 존재하는 경우 등에서 정확한 슬라이싱 정보생성에 어려움을 갖는다. 또한 슬라이싱 기준에 변형[4]을 줌으로써 다양한 슬라이싱 정보를 생성하는 연구도 계속 진행되고 있으며 생성된 슬라이싱정보의 적절한 적용[17][11]에 관한 연구도 활발하게 진행되었다.

슬라이싱 정보의 적용 분야는 다음과 같다.

- 디버깅[13]
- 테스트[7]
- 병렬처리[1]
- 통합[5]
- 소프트웨어 안전성[3]
- 프로그램 이해[8]

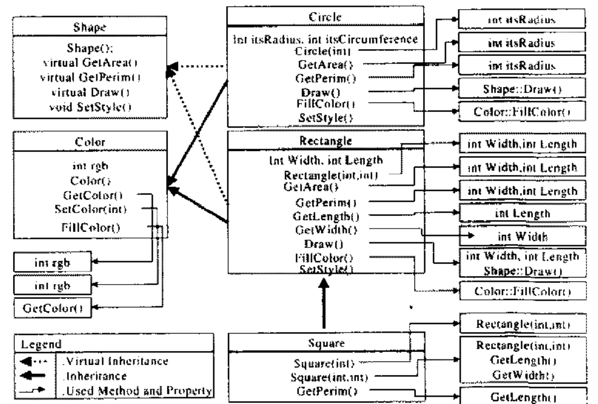
3. 클래스 계층 구조 슬라이싱

클래스 계층 구조 슬라이싱은 고전적인 문장 슬라이싱의 확장으로 클래스간의 의존관계를 객체 지향적인 특성으로 확장함으로써 프로그램에 사용되지 않는 멤버함수 및 멤버데이터를 제거함으로써 프로그램 이해와 디버깅을 쉽게 하고 프로그램의 코드를 최적화 함으로써 수행능력을 향상시킨다. 본 논문에서 제시한 알고리즘은 C++ 클래스 계층 구조와 이 계층 구조를 사용하는 프로그램을 입력받는다. 이 정보를 이용하여 먼저 객체 특징에 맞는 클래스 계층 구조 정보를 추출하여 데이터베이스화하고 이 정보를 기반으로 메인 프로그램에 사용되어지지 않는 멤버데이터와 멤버함수 및 불필요한 상속관계를 제거한다. 본 논문에서 적용하는 클래스 계층 구조 슬라이싱을 이용한 최적화 단계에 대한 전체적인 관점은 4가지로 나누어질 수 있다.

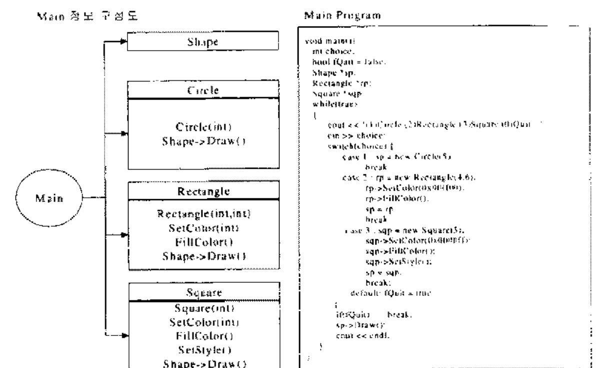
① 클래스 계층 구조 정보로부터 클래스 정보를 추출하는 단계
② 메인 프로그램으로부터, 사용되어지는 멤버함수 및 멤버데이터 정보를 추출하는 단계
③ 이 메인 프로그램 정보 테이블을 이용한 클래스 정보 테이블의 사용여부 검색 및 등록 단계
④ 등록된 정보 테이블을 이용한 최적화 단계

이러한 구성단계는 좀더 구조적인 프로그램을 가능하게 한다. 여기에서는 각 정보 추출 단계를 위한 적용방법을 설명하고 정보추출 단계에서 구성된 테이블을 기반으로 몇 가지의 알고리즘을 적용시킴으로써 최적화 시스템을 구현한다.

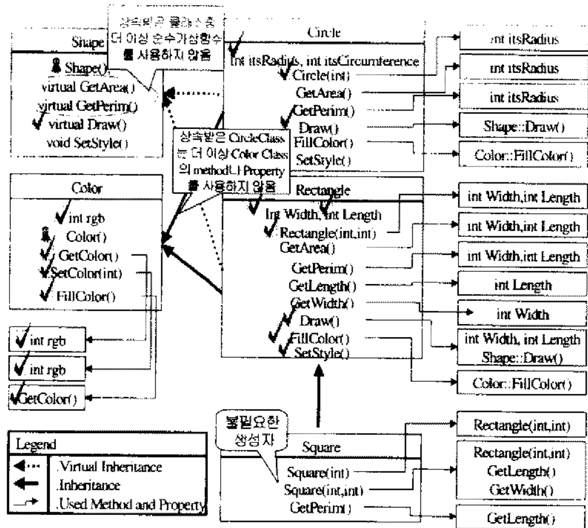
이제 슬라이싱과 최적화에 적용시킬 한가지 예를 통해 객체 지향 언어의 특징을 가진 언어를 대상으로 슬라이싱과 최적화 과정에 필요한 요소들을 적용시켜 나갈 것이다. 즉 상속과 가상함수 사용 등과 같은 형태를 통한 정보테이블 생성 방법을 제시할 것이고, 이 정보 테이블을 통한 불필요한 생성자, 상속관계, 순수 가상함수 처리에 대한 알고리즘을 제시할 것이다.



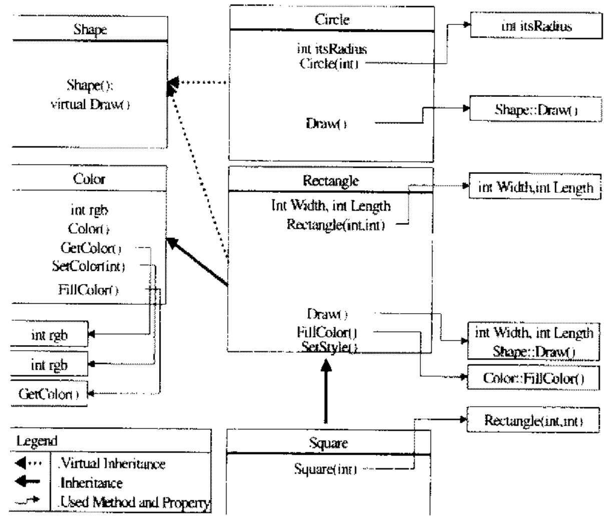
(그림 2) 클래스 구성도



(그림 3) 메인 프로그램 구성도 및 프로그램



(그림 4) 슬라이싱 및 최적화 대상선정



(그림 5) 최적화 후 클래스 구성도

(그림 2)부터 (그림 5)까지는 클래스 계층과 메인 프로그램를 이용하여 클래스 계층에 존재하는 불필요한 멤버함수 및 멤버데이터, 상속관계를 제거하는 일련의 과정을 도식화한 것이다. (그림 2)는 클래스 계층 구조도이다. 각 클래스간의 상속관계와 클래스에 존재하는 멤버함수 및 멤버데이터의 초기 관계를 보여주고 있다. 다음 (그림 3)은 (그림 2)의 클래스 계층 구조를 이용하여 프로그램 되어지는 메인 프로그램을 보여주고 있다. 이때 이 메인 프로그램은 클래스 계층 구조의 클래스 특성 모두를 사용하지 않는 형태이다. 그러므로 메인 프로그램을 기준으로 슬라이싱을 통해 불필요한 멤버함수 및 멤버데이터 그리고 상속관계를 제거할 필요성이 존재한다. (그림 4)는 메인 프로그램의 사용된 클래스와 멤버함수를 기준으로 C++의 특성을 고려하여 사용되어지는 멤버함수, 멤버데이터 및 상속관계를 검색하는 과정을 도식화하였다. 여기에서 사용된 멤버함수를 검색할 수 있고 멤버함수사이에서 사용되어지는 멤버데이터를 검색할 수 있다. 또한 검색된 데이터를 기반으로 불필요한 상속관계 및 순수가상함수, 생성자 등의 검색이 가능하다. (그림 5)는 이러한 불필요한 요소를 제거한 후의 내용을 도식화한 것이다. 코드 시스템 최적화 부분에서, 이러한 과정을 수행하기 위해 객체관리 방법을 제시할 것이고, 객체 관리를 통해 시스템 최적화 과정을 위한 알고리즘을 제시한다.

먼저 여기에서 사용되어지는 프로그램은 아래와 같다.

```
 1 #include <iostream.h>
 2 Class Shape
 3 {public:   Shape(){}
 4            ~Shape(){}
 5            virtual long GetArea()=0;
 6            virtual long GetPerim()=0;
 7            virtual void Draw();
 8            void SetStyle();
 9 };
10 void Shape::Draw()
11 {cout << "Abstract drawing mechanism!" <<
12 endl;}
13 void Shape::SetStyle()
14 {cout << "Abstract seting Style!"<<endl;}
15 class Color
16 {private: int rgb;
17 public:   Color(){}
18           ~Color(){}
19           int GetColor() { return rgb;}
20           void SetColor(int in_rgb)
21           { rgb=in_rgb;}
22           virtual void FillColor();
23 };
24 void Color::FillColor()
25 {cout << "Abstract FillColor mechanism!,
26 Fill Color = "<<GetColor()<< endl;}
27 class Circle:public virtual Shape,public Color
28 {private: int itsRadius;
29           int itsCircumference;
30 public:   Circle(int radius)
31           {itsRadius = radius;   }
32           ~Circle(){}
33           long GetArea()
34           { return 3*itsRadius*itsRadius;}
35           long GetPerim()
36           {return 9*itsRadius;}
37           void Draw();
38           void FillColor();
39           void SetStyle();
40 };
41 void Circle::Draw()
42 {
43    cout << "Circle drawing here!" << endl;
44           Shape::Draw();
45 }
46 void Circle::SetStyle()
47 {
48  cout << "Setting Style for Circle !" << endl;
49 }
50 void Circle::FillColor()
```

```
51 {
52   cout << "Circle Filling  here!" << endl;
53   Color::FillColor();
54 }
55 class Rectangle : public virtual Shape,public Color
56 {
57 private:  int itsWidth;
58           int itsLength;
59 public:   Rectangle(int len,int width)
60           {         itsLength = len;
61                     itsWidth = width;
62           }
63           ~Rectangle(){}
64           long GetArea()
65           {return itsLength * itsWidth;}
66           long GetPerim()
67           {return 2*itsLength+2*itsWidth;}
68           virtual int GetLength()
69                   { return itsLength;}
70           virtual int GetWidth(){ return itsWidth;}
71           void Draw();
72           void SetStyle();
73           void FillColor();
74 };
75 void Rectangle::Draw()
76 {
77           for(int i=0;i<itsLength;i++)
78           {
79                     for(int j=0;j<itsWidth;j++)
80                               cout << "X";
81                     cout << endl;
82           }
83           Shape::Draw();
84 }
85 void Rectangle::SetStyle()
86 {
87   cout << "Setting Style for Rectangle !" << endl;
88 }
89 void Rectangle::FillColor()
90 {
91   cout << "Rectangle Filling Color !" << endl;
92   Color::FillColor();
93 }
94 class Square:public Rectangle
95 { public:
96           Square(int len){ Rectangle(len,len);}
97           Square(int len,int width);
98           ~Square(){}
99           long GetPerim(){ return 4*GetLength();}
100 };
101 Square::Square(int len,int width):Rectangle(len,width)
102 {
103   if(GetLength()!=GetWidth())
104   cout << "Error, not a square... ?" << endl;
105 }
```

(그림 6) 최적화 대상 클래스 프로그램

이 클래스에서 최적화 대상은 첫째, (그림 3)의 메인 프로그램에서 사용되지 않는 일반 멤버함수와 멤버데이터(라인번호 8, 13, 14, 29, 33, 34, 35, 36, 38, 39, 46, 47, 48, 49, 50, 51, 52, 53, 54, 64, 65, 66, 67, 68, 69, 70, 99) 둘째, 불필요한 상속관계 제거 부분으로 상속받은 클래스가 더 이상 부모 클래스의 특징을 이용하지 않을 경우 제거(라인번호-27 public Color) 셋째, 불필요한 생성자 제거 부분으로 메인 프로그램에서 사용되어지지 않는 생성자는 제거(라인번호-97, 101, 102, 103, 104, 105) 넷째, 상속 클래스에서 더 이상 사용되어지지 않는 부모 클래스의 순수 가상함수 제거 부분(라인번호-5, 6)으로 나눌 수 있다. (그림 4)에서 보여주는 것처럼 표시가 된 부분을 제외한 나머지 부분들이 제거 대상 부분들이 된다. 이 과정은 클래스의 정보 데이터베이스와 메인 정보 데이터베이스를 이용하고 C++의 언어적 특징을 기반으로 수행되어진다. (그림 4)과 같은 최적화 대상을 제거한 후 구성된 프로그램은 다음과 같다.

```
include <iostream.h>
class Shape
{public:    Shape(){}
            ~Shape(){}
            virtual void Draw();
};
void Shape::Draw()
{cout << "Abstract drawing mechanism!" << endl;}
class Color
{private:   int rgb;
public:     Color(){}
            ~Color(){}
            int GetColor() { return rgb;}
            void SetColor(int in_rgb){ rgb=in_rgb;}
            virtual void FillColor();
};
void Color::FillColor()
{cout << "Abstract FillColor mechanism!,
 Fill Color = "<<GetColor()<< endl;}
class Circle:public virtual Shape
{private:   int itsRadius;
public:     Circle(int radius)
            {itsRadius = radius;   }
            ~Circle(){}
            void Draw();
};
void Circle::Draw()
{
   cout << "Circle drawing here!" << endl;
            Shape::Draw();
}
class Rectangle : public virtual Shape,public Color
{
private:    int itsWidth;
            int itsLength;
public:     Rectangle(int len,int width)
            {         itsLength = len;
                      itsWidth = width;
            }
            ~Rectangle(){}
            void Draw();
            void SetStyle();
            void FillColor();
};

void Rectangle::Draw()
{
            for(int i=0;i<itsLength;i++)
            {
                      for(int j=0;j<itsWidth;j++)
                                cout << "X";
                      cout << endl;
            }
            Shape::Draw();
}
void Circle::Draw()
{
   cout << "Circle drawing here!" << endl;
            Shape::Draw();
}
```

```
class Rectangle : public virtual Shape,public Color
{
private:    int itsWidth;
            int itsLength;
public:     Rectangle(int len,int width)
            {          itsLength = len;
                       itsWidth = width;
            }
            ~Rectangle(){}
            void Draw();
            void SetStyle();
            void FillColor();
};
void Rectangle::Draw()
{
   for(int i=0;i<itsLength;i++)
   {
            for(int j=0;j<itsWidth;j++)
                       cout << "X";
                       cout << endl;
   }
   Shape::Draw();
}

void Rectangle::SetStyle()
{
  cout << "Setting Style for Rectangle !" << endl;
}
void Rectangle::FillColor()
{
  cout << "Rectangle Filling Color !" << endl;
  Color::FillColor();
}
class Square:public Rectangle
{ public:
            Square(int len){ Rectangle(len,len);}
            ~Square(){}
};
```

(그림 7) 최적화된 클래스 프로그램

이제 이 예제를 기반으로 각 과정을 수행하기 위한 객체 관리 및 최적화 시스템을 소개한다.

3.1 코드 최적화 시스템 소개

이 절에서 제시할 전체적 시스템 구조는 C++에 대한 특성인 단일/다중상속, 정적/동적 바인딩, 함수중복/함수재정의, 순수가상함수/가상함수, 생성자 등을 전제로 하여 클래스 최적화를 적용시키고 있다. 여기에서 사용되어지는 클래스 계층 구조의 표기, 조직 및 주어진 클래스의 부객체의 셋(set)과 클래스 멤버의 선택에 관한 정규화는 Rossie and Friedman의 정규화 표기법 [12]을 기반으로 하여 작성된다.

클래스 구조와 서브 객체 표기

r : Class Hierarchy

C : 클래스 이름 셋

M : 멤버 이름의 셋

- Non-virtual
- virtual

S : 클래스 C 사이의 가상클래스 셋

P : 클래스 C 사이의 비 가상 클래스 셋

이 표기법에서 위 클래스 구조는 다음과 같은 형태로 구성된다.

C(r) = {Color,Shape,Circle,Rectangel,Square}

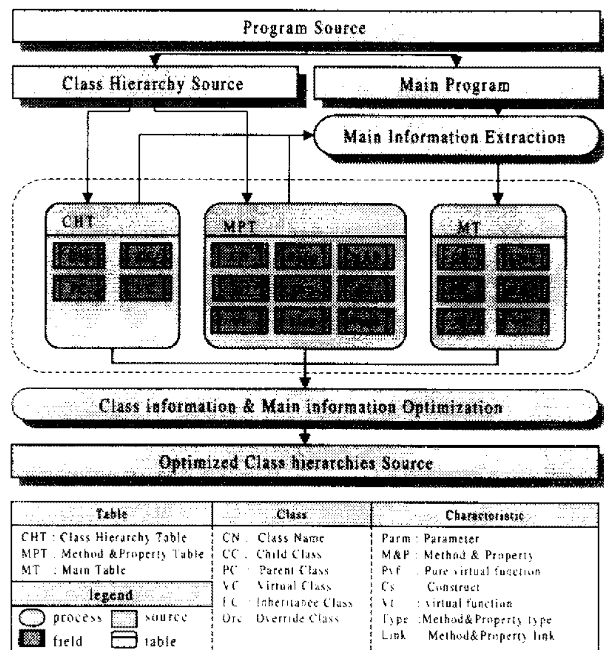
M(r) = {<Color,rgb,non-Virtual>,<Color,GetColor(), Non-virtual>,... <Shape,Shape(),non-Virual>, <Shape, GetArea(),virtual>,...}

S(r) = { <Circle,Shape>,<Rectangle,Shape>}

P(r) = {<Circle,Color>,<Rectangle,Color>,<Square, Rectangle>}

(그림 8)의 최적화 시스템을 위한 객체 관리기 구성은 객체 지향 프로그램을 위한 환경하에서 최적화할 수 있는 독립적인 객체의 정보를 통합하고 관리할 수 있는 기능을 가진다.

객체 관리기는 각 기능의 모듈로써 클래스 상속관계 테이블, 멤버함수 및 멤버데이터 테이블, 메인정보 테이블를 구성하기 위한 각 정보 추출에 대한 모듈과 데이터 등록처리 및 프로그램 최적화 모듈로 구성된다.



(그림 8) 최적화 시스템 구성도

3.2절부터 3.4절까지는 각 단계의 최적화 알고리즘을 적용하기 위해 시스템에 필요한 테이블 구성을 다루고 3.5에서는 구성된 테이블을 기반으로 프로그램 코드의 최적화 과정을 제시한다.

3.2 객체관리

객체 지향 프로그램 환경에서 객체의 정보를 추출하고 관리하는 객체 관리 정보테이블 구성 특징은 다음과 같다.

첫번째는 클래스간의 멤버함수 및 멤버데이터의 관계를 링크형태로 구성함으로써 부모클래스와 조상클래스의 추적이 용이하고, 링크 추적을 통한 사용되어진 멤버함수 및 멤버데이터 추적이 용이하다는 것이다. 이러한 형태의 구성은 C++의 특성 중 함수재정의된 멤버함수 및 멤버데이터의 검색을 용이하게 한다.

두번째는 클래스 정보 구성시 처음 한번의 구축으로 메인 프로그램을 이용한 최적화 수행시 클래스 계층구조 프로그램 코드를 분석하는 대신 구축된 클래스 정보를 이용함으로써 성능 향상을 기할 수 있다.

세번째는 정보 가공 효과를 들 수 있다. 즉 클래스 정보를 데이터 베이스화 함으로써 프로그램 시각화 도구 및 다른 기술에 대한 적용이 가능하다.

객체 관리를 위해 사용되어지는 클래스 정보테이블은 클래스 상속관계 테이블 , 클래스에 사용되는 멤버함수 및 멤버데이터 관계 테이블, 그리고 메인 정보테이블 등 3개의 테이블로 구성된다. 각 테이블의 요소는 위 (그림 6)의 최적화 대상 클래스 프로그램 와 (그림 3)의 메인 프로그램을 이용해 구성하였다.

<표 1>의 클래스 상속관계 테이블은 클래스 계층구조에 대한 상속 정보를 표현하고있으며 이 테이블은 상속관계 제거 알고리즘 및 순수가상함수 처리에 연관되어 사용되어진다.

IT(class) =<Parent Class, Virtual Class, Child Class>

〈표 1〉 클래스 상속관계 테이블

Class Name	Parent Class	Virtual Class	Child Class
Shape	x	x	Circle,Ractangle
Color	x	x	Circle,Rectangle
Circle	Shape,Color	Shape	x
Ractangle	Shape,Color	Shape	Square
Square	Ractangle		x

<표 2>의 멤버함수 및 멤버데이터 테이블은 클래스 계층 구조에 관한 전체적인 구성요소를 표시하는 정보테이블이다. 각 구성요소로는 순수가상함수, 가상함수, 함수재정의, 함수중복, 생성자 등이 있는데, 이것들은 각 멤버함수 및 멤버데이터의 특성들을 담고 있다.

MT<Class, M&P>=<pvf,vf,type,parm,orc,Cs,U,link>

〈표 2〉 멤버함수 및 멤버데이터 테이블

No	Class	M&P	pvf	vf	type	parm	orc	Cs	U	link
1	Shape	Shape()				void		o		
2		GetArea()	yes	yes	long	void				
3		GetPerim()	yes	yes	long	void				
4		Draw()	no	yes	void	void			o	
5		SetStyle()			void	void				
6	Color	rgb			int				o	
7		Color()				void		o		
8		GetColor()			int	void			o	6
9		SetColor()			int	int			o	6
10		FillColor()	no	yes	void	void			o	8
11	Circle	itsRadius			int				o	
12		itsCircum...			int					
13		Circle()				int		o	o	11
14		GetArea()			long	void	Shape			11
15		GetPerim()			long	void	Shape			11
16		Draw()			void	void	Shape		o	4
17		FillColor()			void	void	Color			10
18		SetStyle()			void	void	Shape			
19	Rectangle	itsWidth			int				o	
20		itsLength			int				o	
21		Rectangle()				int,int		o	o	19,20
22		GetArea()			long	void	Shape			19,20
23		GetPerim()			long	void	Shape			19,20
24		GetLength()	no	yes	int	void				20
25		GetWidth()	no	yes	int	void				19
26		Draw()			void	void	Shape		o	19,20,4
27		FillColor()			void	void	Color		o	10
28		SetStyle()			void	void	Shape		o	
29	Square	Square()				int		o	o	21
30		Square()				int,int		o		21,24,25
31		GetPerim()			long	void	Shape			24

Legend	
M&P	멤버함수 및 멤버데이터
pvf	pure virtual function
vf	virtual function
type	value = type, method=return type
parm	argument
orc	overriding Class
Cs	Constructor
U	Used Medthod and Property
link	사용되어지는 멤버함수 및 멤버데이터 관계

<표 3>의 메인 정보 테이블은 사용된 내용에 대한 정보저장 테이블이다. 이 메인 정보 테이블을 이용하여 클래스 계층구조에 포함된 멤버함수 및 멤버데이타

의 사용여부를 확인할 수 있다.

M_MT<Class, M&P&Pointer> = <virtual,inheritance, PClass,link>

〈표 3〉 메인 정보 테이블

Class	M&P&Pointer	Virtual	Inheritance	PClass	Link
Shape	*sp			Circle	
Shape	*sp			Rectangle	
Shape	*sp			Square	
Circle	Circle(5)				13
Rectangle	Rectangle(4,6)				21
Rectangle	SetColor(0x00ff00)		Color		9
Rectangle	FillColor()				27
Square	Square(5)				29
Square	SetColor(0x0000ff)		Color		9
Square	FillColor()		Rectangle		27
Square	SetStyle()		Rectangle		28
Shape	Draw()	Circle Rectangle Square	 Rectangle		16 26 26

Legend	
Class	Class Name
M&P&Pointer	Method and Property and Pointer
Virtual	가상함수일 경우 호출된 클래스 이름
Inheritance	함수가 존재하는 상위 클래스 이름
PClass	Point되어진 클래스 이름
Link	Method&Property Table에 존재하는 해당함수의 링크

3.3 클래스 계층정보로부터 클래스 정보를 추출하는 단계

다음 알고리즘 #1은 클래스 계층 구조 프로그램의 내용을 이용해, 클래스 상속정보 테이블과, 클래스 멤버함수 및 멤버데이터 테이블 등록과 관련된 형식을 담고 있다. 다음 과정을 통해 클래스 계층 구조와 관련된 테이블이 구성된다.

<#1 클래스 계층정보로부터 클래스 정보 추출 알고리즘>

```
void ClassInfomationExtractor(pBuf)
{
    while(pBuf != NULL)
    {
        info = Search_ClassInfomation(pBuf);
        if(info.attribute == CLASS_INFO)
        {
         Add_Table(info.class_inheritance_table);
         continue;
        }
        Add_Table(info,method_Property_table);
        if(info.attribute != METHOD) continue;
        if(info.pure_virtual_function) continue;
        overrideClass
            DoSearchAncestor(info.Class,info.Method);
        if(overrideClass != NULL)
         Append_Table(overrideClass,info.Class,info.Method);
        MethodBody(pBuf,info);
    }
}
```

이 구성에서 클래스 계층 구조를 입력받아 정보를 검색하여 클래스 상속 정보테이블과 멤버함수 및 멤버데이터 테이블에 정보를 등록한다. 또한 검색된 정보가 순수 가상함수가 아닌 일반 멤버함수일 경우 함수 재정의된 정보가 있는지를 검색하여 정보를 등록하고, 그 함수의 멤버함수 바디(body) 부분의 정보를 이용하여 클래스에서 사용되고 있는 관계를 링크 형태로 구성하여 테이블에 등록한다. 구성된 정보 테이블 형태는 <표 1>과 <표 2>의 형태를 가진다. 이 과정에서는 멤버함수 및 멤버데이터 테이블의 Used(U) 필드는 채워지지 않는다.

3.4 메인 정보추출 및 사용된 멤버함수, 멤버데이터 등록 처리

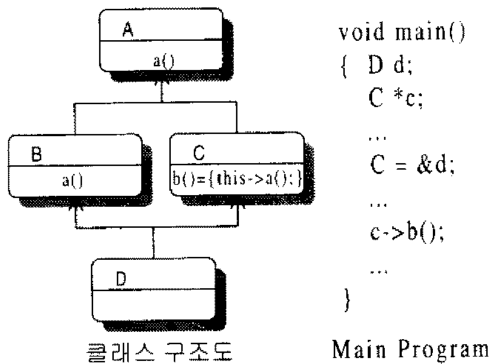
메인 프로그램의 정보 추출 단계는, 프로그램에서 사용되어지고 있는 클래스와 관련된 멤버함수 및 멤버데이터에 대한 정보를 메인 프로그램 테이블에 등록하고 추출된 정보를 이용하여 멤버함수 및 멤버데이터 테이블에 멤버함수 및 멤버데이터의 사용여부를 등록한다. 즉, 이때 <표 2> 멤버함수 및 멤버데이터 테이블의 Used(U) 필드가 설정된다. 이러한 과정의 알고리즘 #2는 다음과 같다.

<#2 메인 정보 추출 및 사용된 멤버함수, 멤버데이터 등록 알고리즘>

```
void MainInfomationExtractor(mainBuf)
{
    while(mainBuf != NULL)
    {
            info = Search ClassInfomation(pBuf);
            Add Table(info,main_table);
            if(info.attribute != METHOD) continue;
            do_Search_Method_Property_Table
                        (info,&className,&Link,&Virtual);
            Append_Table(info,className,Link,Virtual);
            if(Virtual)
            {
              DoVirtualMethod(info,&className,&Link);
              Append_Table(info,className,Link);
```

```
        }
    }
    while(main table != NULL)
    {
            Link = Get_Main_Table_Link();
            do_Insert_Used_Filed(Link,method_property table);
    }
}
```

이 구성에서 메인 프로그램을 입력받아 정보를 검색하여 포인터정보, 멤버함수와 멤버데이터에 대한 정보를 등록한다. 검색된 정보가 멤버함수일 경우 어느 클래스의 멤버함수에 사용되는지를 멤버함수 및 멤버데이터 테이블을 이용하여 확인한 후 그와 관련된 링크정보를 메인 테이블에 등록한다. 만약 사용된 멤버함수가 가상함수일 경우 동적 바인딩을 고려하여 사용클래스와 링크정보를 얻어와서 메인정보 테이블에 등록한다. 이러한 메인 테이블 구성이 완성되면 메인 테이블의 링크 필드 정보를 이용하여 클래스 계층 구조의 정보를 담고있는 멤버함수 및 멤버데이터 테이블의 Used(U) 필드를 등록한다. 이 단계를 수행하면 <표 3>과 같은 테이블 형태가 구성되고, <표 2>의 Used(U) 필드가 설정된다. 여기에서 처리되는 가상함수 동적 바인딩 처리에 대한 좀더 구체적인 접근방법을 알아보기 위해 예제를 통해 수식으로 표현하면 다음과 같다.



(그림 9) 가상함수 처리 예제

이 예제에서 a()가 가상함수일 때 이 a()을 처리하기 위해 사용되는 멤버함수를 확인하는 과정은 다음과 같은 과정을 거친다.

① 초기선언된 클래스와 연관된 클래스 정보를 추출
   $\Sigma(r,D)=\{[D,D],[D,D.B],[D,D.C],[D,D.B.A],[D,D.C.A]\}$
② 호출된 함수 c->b()에서 호출되는 this->a()의 원래의 타입을 확인한다.
   여기에서는 [D,D]가 여기에 속한다.
③ 원타입의 서브 객체 셋에서 a()를 포함하고 있는 클래스를 확인한다.
   a()함수 포함 클래스는 [D,D.B],[D,D.B.A],[D,D.C.A]이다.
④ 서브 객체 셋에서 찾은 클래스들 중 이 가상클래스의 특성에 따라 원 클래스와 가장 가까운 클래스의 멤버함수가 사용되어진다는 것을 알 수 있다.
   여기에서는 [D,D.B]에 존재하는 a()라는 멤버함수가 사용된다.

3.5 등록된 정보 테이블을 이용한 최적화 단계

이 단계는 전 단계에서 구축된 테이블을 이용하여 프로그램 코드를 최적화 시키는 단계로, 사용되지 않는 멤버함수, 멤버데이터 그리고 상속관계를 C++의 특성에 맞도록 제거하는 단계이다. 여기에서는 최적화 과정에 대한 전체적 알고리즘 #3은 다음과 같다.

<#3 최적화 과정에 대한 알고리즘>

```
void ClassOptimize(pBuf)
{
    CString deleteIndex,resultBuf;
    pSet = &method_property_table; // 메소드 및 프로퍼티 테이블
    pcSet = &class_table; // 클래스 상속관계 테이블
  do{
       if(!pSet->Cs && !pSet->Pvf)
         deleteIndex += do_General_Method_Property_Delete();
       else if(pSet->Cs)
              deleteIndex += do_Constract_Method_Delete();
       else
            deleteIndex += do_Pure_Virtual_Delete();
       pSet->MoveNext();
     }while(!pSet->IsEOF());
  do{
       deleteIndex += do_Inheritance_Delete();
     }while(!pcSet->IsEOF();
  resultBuf = Action_Source_Code_Delete(pBuf,deleteIndex);
}
```

이 과정에서는 멤버함수 및 멤버데이터 정보테이블에서 읽어들인 테이블의 레코드를 분석하여 분석된 데이터가 생성자인지 순수가상함수인지 아니면 그 이외의 데이터인지에 따라 각각 생성자 제거 알고리즘(do_Constract_Method_Delete()), 순수가상함수 제거 알고리즘 (do_Pure_Virtual_Delete()) 그리고 일반함수 및 프로퍼티 제거 알고리즘(do_General_Method_Property_Delete())를 통하여 제거 가능한 프로그램 코드의 인덱스 정보를 얻어 실제 프로그램 코드 최적화를 수행한다.

이 최적화 과정에서 사용되어지는 알고리즘은 다음과 같다.

3.5.1 생성자 알고리즘

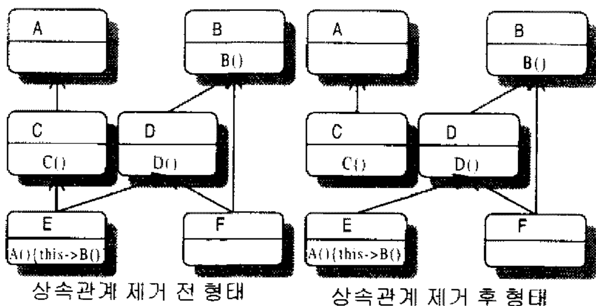
생성자 함수가 각 클래스에 하나 이상 존재해야 한다는 조건을 전제로 한다.

(1) 멤버함수 및 멤버데이터 테이블에서 생성자 함수가 하나 이상 존재하는지를 검색한다.
(2) 생성자함수가 하나 이상일 경우 생성자함수 사용여부를 확인한다. 이 과정은 멤버함수 및 멤버데이터 테이블의 사용여부 필드를 검색하여 확인할 수 있다.
(3) 사용하지 않는 생성자는 제거한다.

3.5.2 클래스 상속관계 알고리즘

(1) 클래스 상속정보테이블에서 각 클래스에 관한 부모 클래스 정보를 얻어온다.
(2) 각 클래스와 관련된 부모클래스가 메인 프로그램에서 사용되고 있는지를 확인하기 위해 멤버함수 및 멤버데이터 테이블에서 그 클래스에 사용되고 있는 클래스 정보를 얻어온다.(DoInheritance-Check)
(3) 멤버함수 및 멤버데이터에서 사용되는 클래스 정보가 상속정보 테이블의 클래스의 부모클래스와 같은지를 검색해 나가면서 해당 클래스의 부모클래스가 사용되고 있는지를 확인한다.(DoInheritance CodeDelete)
(4) 사용이 되지 않는 부모클래스는 프로그램 코드를 분석하여 해당부분을 제거하기 위한 구간정보를 얻어온다.(DoInheritanceDeleteAction)

이 알고리즘을 이해하기 위해 나음과 같은 예제를 생각해 보자



(그림 10) 상속관계 처리 예제

E클래스에서 사용되고 있는 상속관계를 확인하기 위해 다음과 같은 단계를 수행한다.

1. E 클래스의 부모 클래스 정보 검색

   Σ(r, E)=[E, E.C],[E, E.D],[E, E.C.A],[E, E.D.B]

   E의 부모클래스 = [E, E.C],[E, E.D]
2. 클래스 E에서 사용되고 있는 멤버함수 B()가 속한 클래스를 찾는다. 이 정보는 클래스 멤버함수 및 멤버데이터 테이블의 링크정보를 이용해서 찾을 수 있다.

   Method B()가 속한 클래스는 [E, E.D.B] 이다.
3. Method B()가 속한 클래스 [E, E.D.B]가 클래스 E의 어느 부모 클래스에 속하는지를 검색하기 위해 [E, E.D.B]클래스의 자식클래스를 검색한다.

   여기에서 자식클래스는 = {[E, E.D], [F]}이다.
4. 검색된 자식클래스가 E클래스의 부모클래스인지를 확인한다.

여기에서 [E, E.D]가 E클래스의 부모클래스이다. B()가 속한 클래스의 자식클래스가 E의 부모 클래스일때까지 B()가 속한 자식클래스를 재귀적인 호출 과정을 통해 E클래스와 B()이 속한 클래스 사이의 상속관계를 검색한다.

적용 알고리즘 #4는 다음과 같다.

<#4 상속관계 제거 알고리즘>

```
void CRedNoteView::DoInheritanceDelete(CMpInfoDbSet *pSet,CString pBuf)
{
    CClassDbSet* pcSet;
    while(!pcSet->IsEOF())
    {
        if(pcSet->m_ParentClass == "")
            { pcSet->MoveNext();continue; }
        t_ParentClass = pcSet->m_ParentClass;
        ...
        pSet->MoveFirst();
        while(!pSet->IsEOF()){
            if(pcSet->m_ClassName != pSet->m_Class)
                {pSet->MoveNext();continue;}
            if(!pSet->m_Used) {pSet->MoveNext();continue;}
            if(pSet->m_Link == "") {pSet->MoveNext();continue;}
            t_Link = pSet->m_Link;
            long mDbIndex = pSet->GetAbsolutePosition( );
            DoInheritanceCheck(pcSet,pSet,&resultClass);
            pSet->SetAbsolutePosition(mDbIndex);
            pSet->MoveNext();
        }
        long cDbIndex = pcSet->GetAbsolutePosition( );
        DoInheritanceCodeDelete(pcSet,resultClass,useChk);
        pcSet->SetAbsolutePosition(cDbIndex);
        DoInheritanceDeleteAction(pBuf,pcSet,useChk);
        ...
        pcSet->MoveNext();
    }
}
```

resultClass에 각 클래스와 관련된 멤버함수 및 멤버데이터 테이블을 이용하여 사용되고 있는 클래스들의 정보를 구한다. 이 수행 원리는, 먼저 멤버함수 링크정보를 읽어들여 그 링크에 해당하는 클래스가 부모클래스일 경우 사용된 클래스로 등록하고 아닌 경우 다시 재귀적 함수 호출을 하면서, 해당멤버함수에 사용되고 있는 부모클래스 리스트를 검색한다. 이 과정을 통해 사용되지 않는 상속관계를 검색하고 이 정보를 이용하여 실제 프로그램 대상 불필요한 상속관계를 제거한다.

3.5.3 순수가상함수 알고리즘

순수가상함수 제거 루틴은 다음과 같은 과정을 거친다.

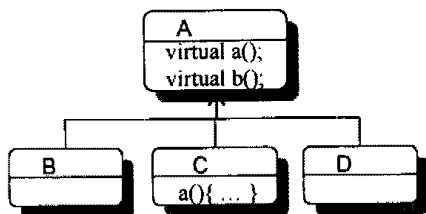
(1) 멤버함수 및 멤버데이터 테이블에서 순수가상함수가 포함되어있는 클래스를 찾는다.

(2) 이 클래스의 자식클래스에서 이 순수가상함수가 사용되고 있는지를 클래스 멤버함수 및 멤버데이터 테이블에서 검색하여 사용유무를 확인한다. (DoPureMethodSearch)

(3) 만약 사용되고 있지 않는 경우에는 해당되는 순수가상함수 정보를 이용하여 프로그램 코드의 제거 부분에 대한 구간정보를 얻어온다.(DoPure-MethodDelectAction)

이 과정은 초기에 자식클래스에 존재했던 순수가상함수 부분이, 메인 프로그램에 의해 상속받은 모든 클래스가 순수가상함수를 사용하지 않을 경우에 발생한다.

이 제거 과정을 수식을 이용해 표현하기 위해 다음과 같은 예제를 생각해 보자.



(그림 11) 순수가상함수 처리 예제

A 클래스에 존재하는 순수가상함수의 사용여부를 확인하기 위해 다음과 같은 과정을 수행한다.

① 클래스 A에서 순수 가상함수 M를 찾는다.
M={<A,a(),virtual>,<A(),b,virual>}이다.

② 이 멤버함수를 포함하는 클래스의 자식클래스 정보를 클래스 상속관계 테이블을 이용하여 얻는다. A의 자식클래스는 {[B],[C],[D]}가 된다.

③ 자식클래스에서 가상함수 a() 와 b()가 존재하는 클래스를 클래스의 멤버함수 및 멤버데이터 테이블을 통해 검색한다. 자식클래스에 포함된 순수가상함수의 멤버함수 M = {<C,a(),non-virtual>}이다.

그러므로 순수가상함수 a()가 클래스[C]에 의해 사용되고 있다는 것을 확인할 수 있다.

적용 알고리즘 #5는 다음과 같다.

<#5 불필요한 순수가상함수 제거 알고리즘>

```
void CRedNoteView::DoPureMethodDelete(CMpInfoDbSet* pSet,CString pBuf)
{
  CString t_Class;
  BOOL isUse;
  pSet->MoveFirst();
  while(!pSet->IsEOF())
  {
     if(!pSet->m_Pvf){pSet->MoveNext();continue;}
     t_Class = pSet->m_Class;
     long mIndex = pSet->GetAbsolutePosition( );
     isUse = DoPureMethodSearch(pSet);
     pSet->SetAbsolutePosition(mIndex);
     if(!isUse)
        DoPureMethodDelectAction(pSet,pBuf);
     pSet->MoveNext();
  }
}
```

불필요한 순수가상함수를 제거하기 위해 먼저 순수가상함수가 포함된 클래스를 상속받은 자식클래스로부터 이 순수가상함수가 사용되어있는지를 검색하고 자식클래스들 중 어느 곳에서 사용하지 않는다면 그 순수 가상함수는 프로그램에서 제거된다.

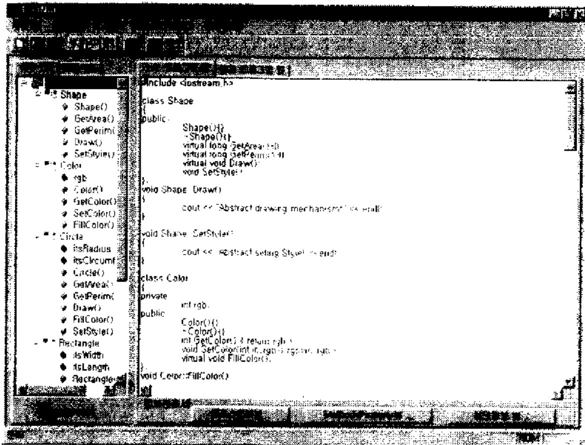
4. 시스템 적용 및 평가

이 적용 예는 클래스 계층 구조와 메인 프로그램의 입력을 받아 구성된 테이블을 기반으로 최적화 과정을 수행하는 실행 화면이다. 구성형태는 대상 프로그램과 결과 프로그램을 탭 형식으로 나누고 왼쪽은 이 프로그램에서 사용한 클래스와 멤버함수 및 멤버데이터를 트리 형태로 구성하였으며 오른쪽은 원시 코드를 표시하도록 하였다.

4.1 적용 예

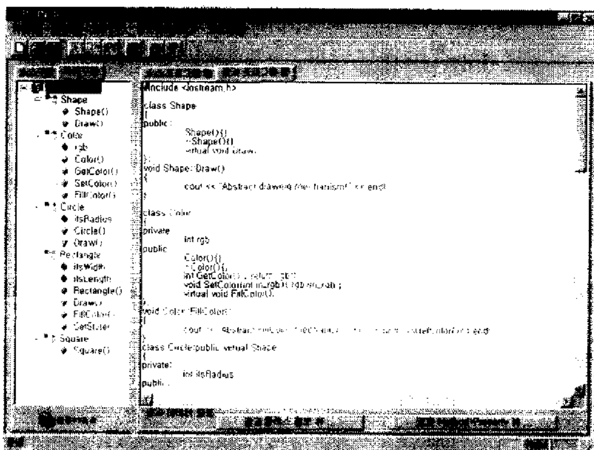
(그림 12)는 최적화 전 초기화면으로 클래스 계층 구조와 메인 프로그램으로 (그림 6) 최적화 대상 클래스 프로그램를 나타낸 것으로 불필요한 멤버함수, 멤

멤버데이터, 상속관계를 가지고 있다. 이 프로그램 코드를 최적화하기 위해 객체관리정보를 데이터베이스화하고 그 정보를 기반으로 최적화를 수행한다.



(그림 12) 최적화 전 초기화면

(그림 13)은 최적화 후 결과 화면으로 객체 관리에 의해 제공되어지는 정보를 이용하여 메인 프로그램의 내용에 최적화된 클래스 계층 구조를 가진 형태를 보여준다. 이러한 최적화는 정보 테이블과 C++의 언어적 특성을 고려하여 최적화 하였다. 결과적으로 실제 메인 프로그램 적용에 필요한 클래스 계층구조의 변화는 불필요한 멤버함수, 멤버데이터, 상속관계가 제거된 형태로 나타난다. (그림 13)은 (그림 7)과 같은 형태로 최적화 되어 나타난다.



(그림 13) 최적화 후 결과 화면

4.2 기존의 시스템 비교

시스템 비교는 C언어기반 슬라이싱과 기존의 C++언어 슬라이싱, 그리고 본 논문에서 제시한 슬라이싱의 특성을 중심으로 구분한다. C 언어로서 프로그램 방법론의 변화는 분석도구의 많은 변화를 가져왔으며 이러한 기존의 C++ 슬라이싱에 정보단위 모듈 구성을 통한 시스템 구축의 확장이라는 측면에서 고려해 보면 다음과 같다.

〈표 4〉 기존의 시스템과 비교

비교대상	C 언어기반 슬라이싱	기존의 C++슬라이싱	본 논문에서 제시한 슬라이싱
부품의 성격	함수	클래스	클래스
슬라이싱 방법	변수 기준 슬라이싱	클래스특성 단위 슬라이싱	클래스 특성단위 정보추출 슬라이싱과 추출정보 기반 최적화 슬라이싱으로 모듈화
재사용 방법	함수단위 재사용	클래스 단위 재사용	클래스 단위 재사용 클래스 정보 재사용
클래스 구조 자동수정	없음	자동화되지 않음	테이블정보를 이용한 빠른 수정 가능
프로그램 분석 방법	특정변수 기준 프로그램 분석	클래스 구성요소로 프로그램 분석	클래스 정보 테이블을 이용한 체계적 분석

5. 결론 및 향후 과제

소프트웨어에 대한 일반 사용자의 요구를 충족시키기 위하여 많은 기능을 가진 방대한 소프트웨어가 개발되고 있다. 그러나 클래스 계층 구조를 이용한 프로그램 개발이 많아지고 자동화된 클래스 사용증가는 불필요한 프로그램 코드, 컴파일 및 로딩 시간의 증가를 가져오며 개발자의 프로그램 코드 분석의 어려움이 가중시키는 요인이 되어왔다. 이러한 상황에서 사용되지 않는 코드의 삭제를 통한 프로그램의 최적화의 필요성은 더욱 증가되었다. 최적화를 통해 개발자는 더 작은 프로그램 코드를 통한 프로그램 분석 및 시스템 성능 향상을 얻게 된다. 본 논문에서 제시한 최적화 기법은 링크형태의 클래스 테이블 구성을 통해 보다 직관적인 클래스 정보를 얻어올 수 있도록 하였고 이 정보테이블을 통해 실제 시스템에 적용가능 예를 제시하였다. 향후 과제로는

첫째, 추출 정보를 이용한 불필요한 멤버함수 및 멤버데이터의 제거 알고리즘을 위한 정보 추출과정에 자동화가 필요하다.

둘째, 알고리즘의 복잡성 때문에 C++의 전체적인 문법은 다루지 않았다. 향후 이 과정에 대한 확장된 알고리즘이 이루어져야 할 것이다.

셋째, 현재 C++ 언어를 대상으로 구현되어 있으며 이런 과정을 통해 분산언어를 적용할 수 있는 방법에 대한 연구 및 객관성에 대한 연구가 더 필요하다.

참 고 문 헌

[1] J-D. Choi, B. Miller, and P. Netzer, "Techniques for Debugging Parallel Programs with Flowback Analysis," Technical Report 786, University of Wisconsin-Madison, Aug., 1998.

[2] K. B. Gallagher and J. R. Lyle, "Using Program Slicing in Software Maintenance," IEEE Transaction on Software Engineering, Vol.17, No.8, pp.751-761, Aug., 1991.

[3] K. B. Gallagher and J. R. Lyle, "Program Slicing and Software Safety," Proceeding of '93 COMPASS, pp.71-80, June, 1993.

[4] R. Gupta, M. Harrold, and M. Soffa, "An Approach to Gegression Testion Using Slicing," Conf. Software Maintenance, pp.299-308, 1992.

[5] S. Horwitz, J. Prins, and T. Reps, "Intergrating Non-Interfering Versions of Programs," ACM Transactions on Programming Languages and Systems, 11(3):345-387, July., 1989.

[6] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," ACM Transaction on Programming Languages and Systems, Vol.12, No.1, pp.26-60, 1990.

[7] M. Kamkar, P. Fritzson, and N. Shahmehri, "Interprocedural Dynamic Slicing Applied to Interprocedural Data Flow Testing," In Proceedings of the Conference on Software Maintenance-93, pp.386-395, 1993.

[8] F. Lanubile, and G. Visaggio, "Extracting Reusable Functions by Program Slicing," Submitted to IEEE Transactions on Software Engineering, 1992.

[9] L. Lasen, and M. J. Harrold, "Slicing Object-Oriented Software," Proceedings of the 18th International Conference on Software Engineering, pp.495-505, Mar., 1996.

[10] N. Nunt, "Performance testing C++ code," Journal of Object Oriented Programming, January, 1996.

[11] K. J. Ottentein, and Linda M. Ottentein, "The Program Dependence Graph in a Software Development Environment," Proc. of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, Pittaburgh, Pennsilvania, April, 1984.

[12] J. G. Rossie, and D. P. Friendman, "An Algebraic Semantics of Subobjects," Proceeding of '95 OOPSLA, 1995.

[13] T. Shimomura, "The Program Slicing Technique and Its Application to Testing, Debugging and Maintenance," Journal of IPS of Japan, 9(9):1078-1086, Sep., 1992.

[14] F. Tip, Jong-Deok Choi, John Field, and G. Ramalingam, "Slicing Class Hierarchies in C++," Proceeding of '96 OOPSLA, 1996.

[15] J. Zhao, J. Cheng, and K. Ushijima, "Static Slicing of Concurrent Object-Oriented Programs," Proceedings of '96 COMSAC, 1996.

[16] M. Weiser, "Program Slicing," IEEE Transaction on Software Engineering, Vol.10, No.4, pp.246-357, July, 1982.

[17] M. Weiser, "Programmers use Slices when Debugging," Communications of the ACM, pp.446-452, July, 1982.

[18] 김희천, "객체지향 프로그램의 종속성 모델과 프로그램 슬라이싱을 이용한 클래스 테스팅 방법", 서울대학교, 박사학위 논문, 1997.

김 운 용

e-mail : wykim@explore.kwangwoon.ac.kr

1996년 독학사 전자계산학과(이학사)

1999년 광운대학교 전산대학원 전자계산학과(이학석사)

관심분야 : 객체지향 프로그래밍 언어, 객체지향 분산 컴퓨팅 객체지향 분석 및 설계

정 계 동

e-mail : kdjung@cs.kwangwoon.ac.kr

1985년 광운대학교 전자계산학과 (이학사)

1992년 광운대학교 산업대학원 전자계산학과(이학석사)

1992년~현재 광운대학교 전자계산학과 박사과정

관심분야 : 객체지향 프로그래밍 언어 및 데이터베이스, 병렬처리 프로그래밍 언어, 객체지향 분산 컴퓨팅

최 영 근

e-mail : ygchoi@daisy.kwangwoon.ac.kr

1980년 서울대학교 사범대학 수학교육과(이학사)

1982년 서울대학교 계산통계학과 (이학석사)

1989년 서울대학교 계산통계학과 (이학박사)

1983년~현재 광운대학교 전자계산학과 교수

1997년~현재 광운대학교 전산정보원 원장

관심분야 : 병렬 컴파일러, 병렬 프로그래밍 언어, 객체지향 프로그래밍언어, 객체지향 분산 컴퓨팅