□ 기술해설 □

객체지향 재사용과 CASE

동국대학교 최은만*
한국전자통신연구소 김진석**

● 목 차 ●

1. 서 론
2. 재사용 라이브러리
3. 객체지향 패러다임과 재사용
3.1 상속
3.2 결합
3.3 파라미터화 타입
3.4 위임
4. 프레임웍과 디자인 패턴
4.1 프레임웍
4.2 디자인 패턴
5. 객체지향 재사용을 위한 CASE 및 환경
5.1 부품 제작 도구
5.2 부품 이용 도구
6. 결 론

1. 서 론

객체지향 기술의 등장으로 소프트웨어의 재사용 기술은 새로운 기술적 전환기를 맞고 있다. 1983년 재사용 기술의 첫 워크샵 "Reusability in Programming" 이후 10여년 동안의 연구들은 주로 소프트웨어 라이브러리, 추상화 기술, 영역 분석 등의 기술적인 측면과 재사용 프로세스 등의 관리적인 측면을 다루고 있다. 제시된 여러 가지 기술을 적용하여 발표된 Raytheon, GTE, Hitachi, Toshiba, AT&T, HP 등의 재사용 결과를 보면 60% 이상의 높은 재사용율을 보이고 있다[1]. 그러나 이러한 성공적인 재사용은 기술 측면에서보다는 체계적인 관리에 의하여 이루어진 것으로 분석된다. 라이브러리 시스템, 분류 기술, 재사용 부품의 구성, 재사용 지원 환경 등의 기술이 계속 연구되었으나 프로그래밍 패러다임 안에 재사용 개념이 본격적으로 포함되어 있지 않아 한계를 보여 왔다.

객체는 블랙 박스로 재사용할 수도 있고 부분적으로 수정하여 화이트 박스로 재사용할 수도 있는 독특한 특성을 가진 재사용 단위이다. 절차중심 프로그램의 함수 라이브러리처럼 그 내용에 관계없이 인터페이스만을 알면 재사용 가능하다. 이미 정의된 객체와 약간의 차이를 보이는 객체는 기존의 정의를 수정하여 재사용할 수도 있다. 또한 객체지향 프로그램은 자료의 변형으로 영향받는 부분이 적어 재사용 가능성이 많아진다.

객체지향 패러다임에서는 코드 수준의 재사용뿐만 아니라 소프트웨어 구조, 즉 프레임웍이나 디자인 패턴 형태로 재사용되는 경우가 많다. 한정된 응용 분야의 소프트웨어들이 유사한 소프트웨어 구조를 나타내는데 특정 응용에서 보편적으로 적용될 수 있는 시스템의 구성 형태를 프레임웍이라 한다. 객체지향 재사용에서는 프레임웍 안에 재사용 부품들을 결합하여 시스템을 구성한다. 설계에서 자주 반복되는 객체의 구조나 상호작용을 디자인 패턴이라고 한다. 객체지향 프로그래밍에서는 디자인 패턴, 즉 객체의 구조나 객체 사이의 동작이 반복된다.

*종신회원
**정 회 원

객체지향 기술에서 재사용에 의한 개발 과정에는 클래스 라이브러리, 클래스 브라우저, 객체를 파악할 수 있는 디버거, 편집기와 버전제어 시스템, Make 기능들을 통합한 환경이 필요하다. 특히 재사용을 지원하는 CASE(Computer Aided Software Engineering) 도구는 부품을 찾아 시스템으로 합성할 수 있는 기능이 필요하다.

이 논문에서는 객체지향 패러다임에서의 재사용 기술에 대하여 CASE 도구를 중심으로 고찰해 본다.

2. 재사용 라이브러리

코드 라이브러리의 구성이 재사용을 위한 첫 출발이다. 코드 라이브러리의 구성은 재사용 유형과 목적에 따라 달라진다. 블랙 박스 재사용에서는 주로 플러그-인 형태로 사용되므로 재사용 부품이 무엇을 하는가 만을 고려하여 라이브러리를 구성한다. 그러나 화이트 박스 재사용을 위해 구성하는 라이브러리는 재사용 부품이 어떻게 동작하는가도 고려하여야 한다. 따라서 블랙 박스 라이브러리는 분류와 검색만을 제공하면 되나 화이트 박스 라이브러리는 부품의 설명과 융통성도 고려하여야 한다. 결국 이 문제는 부품의 크기 문제로 귀결된다. 부품을 작게 만들면 화이트 박스 재사용이 용이해지나 분류와 검색에 부담이 되며 부품을 크게 만들면 반대의 현상을 보인다.

블랙 박스 재사용을 위한 라이브러리, 특히 사용자 인터페이스나 수치계산 라이브러리는 재사용의 성공적인 사례이다. 그러나 진정한 재사용을 위해서는 블랙박스나 화이트박스 재사용 모두를 지원하는 라이브러리가 되어야 한다. 특히 객체 라이브러리에서는 필수적이다.

대부분의 객체 라이브러리는 상속 관계에 따라 계층적으로 잘 정리되어 있으며 크게 구분하면 다음과 같은 클래스들을 포함하고 있다 [2].

- 확장이나 변경없이 바로 인스턴스화하여 사용할 수 있는 클래스(concrete class) : 클래스 Date, Time, String 등
- 특수 목적의 클래스 : 예를 들면 패턴매칭, 그래픽 좌표, 실수 계산을 위한 클래스
- 자료구조 클래스 : 클래스 Bag, Array, Set, Linked List, Heap, Stack 등
- 사용자 인터페이스 클래스 : Action button, Scroll-bar, Text-display 등
- 특수 응용 분야 클래스 : 네트워크, 데이타베이스, 통계, 수학, 시뮬레이션 등

표 1은 최근 시장에 나와 있는 클래스 부품 라이브러리들을 발췌한 것이다.

표 1 클래스 라이브러리

클래스 라이브러리	부 품	플랫폼	제작사
Tools.h++ Linpack.h++	date, linked list, stack, queue, regular expression class, B-tree class vector, metrics, statistics, signal processing	Unix IBM-PC	Rogue Ware Software
NetClasses	TCP/IP object transport, distributed programming	Sun	PostMordern Computing Technoloy
BTrv++	classes working with Novell's Btrieve 3.1	IBM-PC	Classic Software
Booch Components	data structure classes(queue, ring, set, graph, tree, list...), tools(searching, pattern matching, coroutine, exception handling)		Rational
Mejin++	statistics, mathematics, simulation	MS-DOS	Network Integrated Services
C++_Views	GUI tools	Windows	CNS Inc,
Common View3	C++ GUI classes	Windows OSF/Motif	Glockenspiel
Zinc	GUI library	IBM-PC	ZINC Software

3. 객체지향 패러다임과 재사용

소프트웨어의 설계와 프로그래밍에는 두 가지 차원이 있다. 응용 문제의 차원과 문제의 해결책에 관한 차원이다. 응용 문제의 구조를 중요시하며 여기에 맞추려는 설계 방법을 응용 중심(application-driven)이라 하며 해결책인 프로그래밍 기법 위주의 설계를 해결책 중심(solution-driven)이라 할 수 있다.

객체지향 프로그래밍을 위한 설계는 다분히 응용 중심이라 할 수 있다. 클래스가 응용 문제를 반영하는 추상적 표현이기 때문이다. 따라서 응용 문제에 적합하게 설계된 객체지향 프로그램은 유사한 응용 문제에 대하여 조금만 수정하여 다시 사용할 수 있다. 즉 객체지향 프로그래밍에서 재사용은 문제 영역과 매우 밀접한 관계를 갖고 있다.

객체지향 프로그래밍에서 재사용하는 방법은 상속(Inheritance)만이 아니다. 객체지향 원시코드의 재사용 방법은 크게 다음 네 가지 방법이 있다.

3.1 상속

C++는 상속이라는 방법으로 이미 선언된 클래스의 자료 선언과 함수 선언을 그대로 사용할 수 있게 한다. 독특한 형태의 제어 형태라 할 수 있으나 본질적으로는 어떤 메소드가 어떤 환경에서 사용되어야 하는지를 분류하는 역할을 한다.

새로 만들어야할 클래스가 이미 정의된 클래스의 일종(IS A type)이라면 Public Inheritance를 사용한다. 예를 들면 Rectangle은 GeometricObject의 일종이다.

```
class GeometricObject {
  Point position;              // center of object
  Point bbox;                  // bounding box
  angle orientation;           // orientation
  CoordinateSystem local;// local coordinates
public :
  void move(Point r);    // abs. translate
  void rotate(double angle) = 0;// abs. angle
  virtual void draw() = 0;// display on screen
  // etc....
};
class Rectangle : public GeometricObject {
  // etc...
};
```

새로운 클래스가 베이스 클래스의 유사(IS LIKE A) 클래스라면 Private Inheritance를 사용한다. 예를 들어 Stack은 List와 유사하다. 따라서 다음과 같이 Private Inheritance를 사용한다.

```
class List {
  ListNode *start;
  int nNodes;
public :
  enum {FIRST = 0, Last = -1};
  List() {start = 0;nNodes = 0;}
  ~List() {empty();}
  int add(char *s);
  char *at(int i);
  int index(char *s);
  char *removeAt(int i);
  int size() {return nNodes;}
;
class Stack : private List {
public :
  int push(char*s) {return add(s);}
  char *pop(){return removeAt(First);}
  int size() {return List::size();}
};
```

또 다른 방법은 취소를 포함한 상속이다. 이 방법을 반대하는 의견도 있지만 C++에서 선택적으로 public 또는 private 클래스를 선언하는 방법으로 사용되고 있다. 예를 들어 OrderedList를 정의하려면 위에서 정의된 List 클래스를 이용한다.

```
class OrderedList : protected List{
  public :
    int add(char *s);// ordered add function
    (new)
    List::size;        // make public List::
```

```
size()
};
```

접근이 제한된 클래스 멤버를 public으로 하여 접근 가능하도록 환원할 수는 있지만 반대로 Public으로 선언된 함수를 Protected로 만들 수는 없다.

3.2 결합

결합(composition)은 여러 가지 관련된 클래스를 모아서 새로운 클래스 객체를 만드는 것을 말한다. 예를 들어 Stack 객체를 List 객체를 이용하여 구성하면 다음과 같이 된다.

```
class Stack{
  List list;
public :
  int push(char *s){
      return list.add(s);
  }
  char *pop(){
      return list.removeAt(List : :First);
  }
  int size(){
      return list.size();
  }
};
```

Private 상속은 결합 개념과 매우 유사하다. 차이점은 상속에서는 상속받는 층이 가급적 짧은 것을 원한다. 또한 결합을 이용하면 같은 타입을 여러 번 결합하여 상속받을 수 있다. 예를 들면 다음과 같이 MakeTarget은 여러 개의 StringList 객체를 결합으로 재사용할 수 있다.

```
class MakeTarget {       // simplified...
  String name;           // target name
  StringList depend;     // dependency list
  StringList product;    // production list
  // etc
};
```

다음과 같은 다중 직접 상속은 불가능하다.

```
// illegal
class MakeTarget : String, StringList,
StringList { // etc
};
```

결국 Private Inheritance는 아주 유사한 클래스인 경우에만 사용하여야 하며 관련 클래스 사이의 관계를 잘 정리하여야 한다. 또 한 가지 중요한 차이점은 Public Inheritance는 베이스 클래스의 Public 인터페이스를 파생된 클래스에서 접근 가능하게 한다는 점이다. 따라서 Public Inheritance의 사용은 파생된 클래스가 베이스 클래스 타입인 경우에만 적합하다. 반면에 결합에 의한 객체 인터페이스는 클래스가 결합된 객체의 멤버 함수를 forward하는 경우에만 포함하는 클래스에서 사용할 수가 있다.

```
class UniqueList {
  List list;
 public :
  int add(char *s){
    if (list.index(s) < 0) // element present?
      return list.add(s);
  }
  int remove(char *&s){// message forward-
ing
    return list.remove(s);
  }
  int size(){return list.size();}
  char *at(int pos){return list.at(pos);}
};       // etc ...
```

메시지 forwarding을 사용하면 결합된 객체의 인터페이스가 변경될 때 캡슐화한 클래스의 인터페이스가 전혀 영향을 받지 않으면서 두 클래스가 강력하게 연결된다. Public 상속은 두 클래스 사이에 느슨한 동적 연결을 가능하게 한다. 위에서 정의한 UniqueList 클래스를 public 상속을 이용하여 구현하면 다음과 같다.

```
class UniqueList : public List{
  public :
    int add(char *s){
```

```
    if (index(s) < 0) // element present?
      return List : : add(s);
  }
};
```

3.3 파라미터화 타입

재사용의 또 다른 방법이 파라미터화 타입이다. C++의 template 클래스 및 함수를 이용한다. 파라미터화 타입은 동일한 원시코드를 다른 클래스나 변수 타입으로 재사용하는 마크로 기능이라고 할 수 있다. 예를 들어 여러 가지 타입(int, Windows, char 등)을 위한 스택을 template으로 구현한다면 다음과 같다.

```
template <class T> class Stack;
template <class T> class Cell {
friend class Stack<T>;
private :
  Cell *next;
  T *rep;
  Cell (T *r, Cell<T> *c):rep(r), next(c){}
};
template <class T> class Stack {
public :
  T *pop();
  T *top() {return rep->rep;}
  void push(T *v) {rep = new Cell<T> (v,
rep);}
  int empty() {return rep == 0;}
  Stack() {rep = 0;}
private :
  Cell<T> *rep;
}
template <class T> T *Stack<T> : : pop(){
  T *ret = rep->rep;
  Cell<T> *c = rep;
  rep = rep->next;
  delete c;
  return ret;
}
```

이러한 형태의 재사용의 장점은 동일한 원시코드로 여러 다른 타입의 변수를 가진 클래스를 만들 수 있다는 것이다. 다만 C++는 template 클래스의 인터페이스 선언과 그 구현을 분리하는 메카니즘을 제공하지 않는다. 따라서 template 클래스를 사용하는 모든 화일이 클래스 멤버 함수의 복사본을 가지고 있어야 한다.

3.4 위임

위임(delegation)은 객체에 대한 관계로 클래스에서 상속의 개념과 같다. 위임이란 다른 클래스의 객체의 포인터로 메시지 포워딩하는 것이라 할 수 있다. 위임의 장점은 함수 단위로 정의되기 때문에 객체 인터페이스를 아주 작은 규모로 재사용할 수 있다는 것이다. 반면에 Public 클래스 상속은 인터페이스의 재사용 규모가 크다.

위임은 객체 포인터를 사용하여 메시지를 포워딩하는 결합과는 다르다. 위임된 객체의 다형성을 제공한다. 따라서 위임하는 클래스의 인스턴스들이 같은 클래스의 위임자를 가질 필요가 없다. 사용자 인터페이스를 설계할 때 위임을 사용하는 예는 다음과 같다.

```
class Window;
class WindowDelegate {
  void windowDidInit(Window *){}
  virtual int windowWillMove(Window *)
  {return 1;}
  virtual int windowWillClase(Window *)
  {return 1;}
};
class Window {
  protected :
  // very complicated definition of a window
  WindowDelegate *delegate;
  public :
  void setDelegate(WindowDelegate *del);
};
class MyController : public WindowDelegate {
int windowWillClose(Window *window){
  // test if window contents modified
  // if so, handle saving window contents
}
};
```

위임 방법의 재사용의 단점은 관련된 클래스

사이의 원시코드 의존성이 높아진다는 것이다. 이는 프로그램의 이해를 어렵게 한다. 위임하는 클래스와 위임받는 클래스 사이의 관계가 느슨한 결합을 이루어 위임 클래스의 인터페이스를 변경하면 위임받는 클래스의 원시코드를 다시 컴파일하여야 하는 불편이 있다.

4. 프레임웍과 디자인 패턴

4.1 프레임웍

객체지향 프로그래밍의 경험에 의하면 원시코드뿐만 아니라 설계를 재사용하는 경우가 많다. 객체지향 소프트웨어의 구조는 여러 개의 클래스가 계층관계를 이루어 결합한 것으로 객체 사이에 메시지를 교환하며 객체 상태를 변화시킨다. 동일한 응용분야의 객체지향 프로그램을 보면 객체의 구조와 결합이 일정한 틀을 유지하고 있는데 이를 프레임웍(framework)이라고 한다. 따라서 프레임웍은 단순히 시스템 안에서의 클래스 계층 구조만을 의미하는 것이 아니라 객체 사이의 상호작용에 대한 모델도 포함한다.

프레임웍의 간단한 예로 통계 자료를 처리하여 테이블과 그래프로 보여주는 응용 문제는 그림 1과 같은 소프트웨어 구조를 갖는다[3]. 그래픽 처리(shape), 자료 구조, 테이블이나 그래프 뷰, 그 밖의 응용 고유의 작업을 담당하는 프레임웍으로 구성된다. 테이블 뷰를 담당하는 프레임웍을 자세히 들여다보면 다음과 같은 객체 상호 작용들이 필요하다.

- 보관된 자료를 읽어 테이블 형태로 보여주는 데 필요한 객체들의 상호작용.
- 자료의 값이 테이블 뷰에서 변경되었을 때 보관된 자료를 변경하는 상호작용.

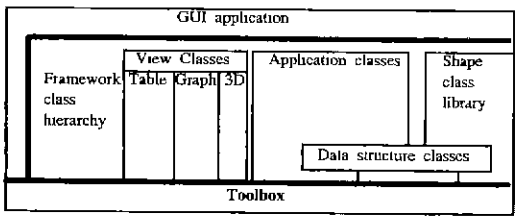

그림 1 프레임웍으로 구성된 객체지향 소프트웨어 구조의 예

- 테이블 뷰에서 자료의 선택이 변경되었을 때 이를 처리하는 상호작용.
- 테이블 뷰에서 필드의 추가와 선택을 처리하는 상호작용.

이 외에도 여러 가지 객체가 결합하고 상호작용하여 테이블 뷰를 담당하는 프레임웍을 이룬다. 프레임웍을 표현하는 방법은 일정하지 않으나 프레임웍을 이루는 객체들의 정적인 계층구조와 클래스 간의 동적인 상호작용을 포함하여야 한다.

4.2 디자인 패턴

객체지향 설계에서 자주 반복되는 구조를 디자인 패턴이라 한다. 객체지향 프로그래밍에서 객체들의 역할과 연결이 유사한 경우가 많다. 예를 들면 그림 2와 같이 여러 객체들이 한 객체의 상태변화에 의존되어 있는 경우 'Observer'라는 패턴을 사용한다. 객체의 상태가 변경되는 것을 관찰하고 있다가 변경이 있으면 의존하는 다른 객체에게 알리는 패턴이다[4].

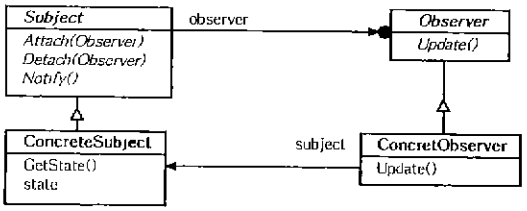

그림 2 디자인 패턴의 예(Observer)

5. 객체지향 재사용을 위한 CASE 및 환경

객체지향 재사용을 위하여는 부품 저장소 중심의 통합된 도구가 제공되어야 한다. 재사용 가능한 클래스 라이브러리를 기초로 클래스 브라우저, 객체를 식별할 수 있는 코드 수준의 디버거, 원시 코드 변경제어 기능, 프로그램 편집기 등이 통합되어 CASE 도구로 구성되어야 한다. 재사용 도구는 크게 재사용 부품을 제작하는 단계에 사용하는 도구(For Reuse)와 재사용 부품을 이용하여 새로운 시스템을 구성하는 단계에 사용하는 도구(With Reuse)로 나눌 수 있다.

5.1 부품 제작 도구

재사용 라이브러리 구성에 필요한 도구는 다음 세 가지이다.

(1) 재공학(Reengineering) 도구

이미 개발된 프로그램에서 재사용 가능한 부품의 추출에 필요하다. 재사용 부품 제작의 초기 투자 비용을 줄이기 위하여 이러한 도구가 필요하다. 일반적인 재공학 도구는 시스템의 기능이나 구조의 향상을 위한 것이지만 여기서는 변경 없이 재사용 가능성이 높은 부품을 찾아내는 데 목적이 있다. REBOOT에서는 자료와 함수를 클러스터링하여 분류한 후 재사용 가능 객체를 추출하는 도구를 제공한다[5]. 소프트웨어 역공학이나 재공학 분야의 연구에서 이 문제를 활발히 다루고 있다[6].

(2) 품질 도구

준비된 부품을 재사용 라이브러리로 보관하기 전에 품질 기준에 적합한지를 검사한다. 여러 가지 메트릭이나 사용된 내역을 바탕으로 부품이 시스템에 관계 없이 호환 가능한지, 쉽게 변경 가능한지, 쉽게 이해되어 재사용할 수 있는지, 새로운 시스템에서 목적에 맞게 사용될 수 있는지 검증할 필요가 있다.

(3) 분류 도구

재사용 부품의 규모가 커진다면 적합한 부품을 효율적으로 찾기 위하여 부품을 적절하게 분류하여야 한다. 일반적인 재사용 부품은 대부분 패싯으로 분류한다. 부품이 어떤 일을 하며(operation), 그 대상은 무엇이며(operation on), 어떤 영역에서 사용되며, 어떤 시스템에서 어떤 언어로 쓰여진 것인지 나타내고 이를 기초로 찾아낸다[7].

객체지향 재사용 부품은 주로 부품의 상속관계로 분류한다. 클래스 부품은 화이트 박스로 재사용하므로 상속된 내용을 찾기 위하여 슈퍼클래스를 찾아보아야 한다. 따라서 상속 계층관계에 따라 분류하여야 한다.

5.2 부품 이용 도구

객체지향 재사용을 위해서 가장 중요한 도구는 클래스 브라우저이다. 단순히 원시코드를 보여주는 것보다 클래스 사이의 계층관계를 보여주고 클래스 정의와 메소드에 대한 정의를 보여 주어야 재사용 대상을 쉽게 찾고 이해할 수 있다. 클래스 구조가 복잡한 경우 브라우저는 가상 메소드(virtual method)가 어디서 소개되었고 파생된 어떤 클래스에서 재정의되었는지 찾아 주어야 한다. 또한 어떤 클래스에 대하여 브라우징할 때 대상 클래스에 정의된 사항만 보여줄 수도 있고 상속에 의한 사항을 모두 보여줄 수도 있다. 브라우저는 특정 메소드가 호출된 부분을 찾아주는 기능도 있어야 한다. 결국 프로그래머는 브라우저를 이용하여 클래스의 특성을 알게 되고 재사용하게 된다.

지금까지 소개된 많은 객체 브라우저들은 Smalltalk 프로그램 환경을 참조한 것이다. 그림 3은 MacBrowse 클래스 브러우저와 같이 화면을 나누어 (1)라이브러리에 있는 클래스, (2)클래스 멤버 함수, (3)데이타 멤버, (4)메소드 정의 등을 보여준다.

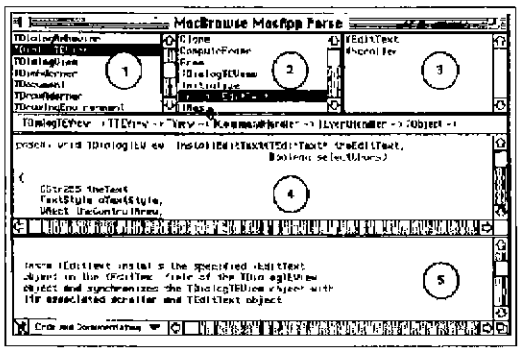

그림 3 MacBrowse 클래스 브라우저

6. 결 론

소프트웨어 재사용을 통하여 소프트웨어 생산성 문제를 해결할 수 있다. 그러나 단순히 재사용 가능한 원시코드를 모아 놓았다고 재사용되는 것은 아니다. 부품을 적당히 분류하고 효율적으로 찾을 수 있는 도구를 제공하고 부품을 이용하여 시스템을 결합할 수 있는 CASE 도구 및 통합 환경을 제공하여야 한다. 또한 개발 집단의 특성을 살린 재사용 프로세스의 관리도 매우 중요하다.

객체지향 패러다임에서의 재사용은 부품의 특성뿐만 아니라 재사용 방법, CASE 도구 구성 등 여러 측면에서 절차 중심 패러다임과는 다른 특징을 보인다. 클래스는 변경하지 않고 그대로 사용하는 부분(closed)과 변경할 수 있는 부분(opened)이 명확히 구분되어 있다. 객체지향 프로그래밍에서는 화이트 박스 재사용이 활발히 이루어진다. 또한 객체지향 재사용을 위한 CASE 도구에서는 클래스 브라우저의 구성이 중요하다.

객체지향 패러다임의 도입은 재사용 기술의 전환을 가져왔다. 원시 코드를 더욱 융통성 있게 재사용할 수 있을 뿐만 아니라 소프트웨어의 구조를 구성하는 프레임웍과 디자인 패턴 등을 재사용하여 설계 단계의 생산성을 높일 수 있다.

참고문헌

[1] J. Neighbors, "An Assessment of Reuse Technology after Ten Years", Proc. 3th International Conference on Software Reuse : Advances in Software Reusability, pp. 6-13, 1994.

[2] N. Gray, Programming with Class, John Wiley & Sons, 1994.

[3] T. Lewis, et. al, Object-Oriented Application Frameworks, Manning, 1995.

[4] E. Gamma, et. al, Design Patterns : Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[5] G. Sindre, R. Conradi, "The REBOOT Approach to Software Reuse", Journal of Systems Software, Vol. 30, pp.201-212, 1995.

[6] J. Ning, A. Engberts, W. Kozaczynski, "Recovering Reusable Components from Legacy Systems by Program Segmentation", Proc. of Working Conference on Reverse Engineering, 1993.

[7] R. Prieto-Diaz, P. Freeman, "Classifying Software for Reusability", IEEE Software, Vol. 4, No. 1, pp.6-16, 1987.

최 은 만

1982 동국대학교 컴퓨터공학과 (학사)
1985 한국과학기술원 전산학과 (석사)
1993 일리노이 공대 전산학과 (박사)
1985~88 한국표준연구소 연구원
1988~89 한국데이타통신(주) 주임연구원
1993~현재 동국대학교 컴퓨터공학과 조교수
관심분야 : 객체지향 소프트웨어공학, 재사용, 역공학, 소프트웨어 유지보수

김 진 석

1982 울산대학교 전자계산학과 졸업(공학사)
1988 동국대학교 전자계산학과 졸업(공학석사)
1982~현재 한국전자통신연구소 정보공학연구실장(책임연구원)
1992 정보처리기술사
관심분야 : 멀티미디어 데이타베이스, 소프트웨어공학, CSCW