소프트웨어공학

12주차-2 : 4부 소프트웨어 설계(1)

12장 소프트웨어 설계 기법

12.1 소프트웨어 설계 개요

12.2 소프트웨어 설계 활동

12.3 설계의 고려사항

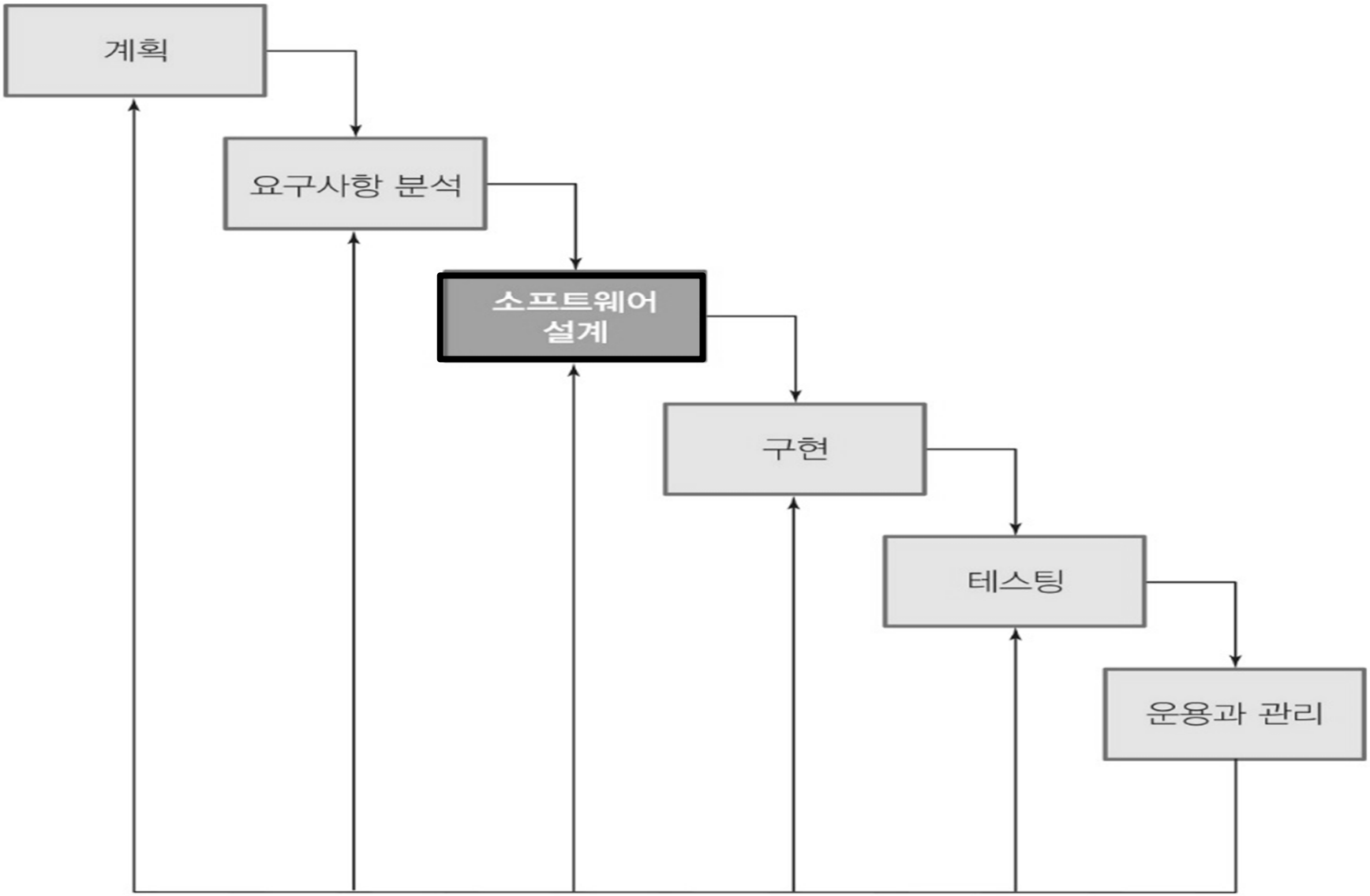
12.4 설계의 품질 요소

kit 국립금오공과대학교
Kumoh National Institute of Technology

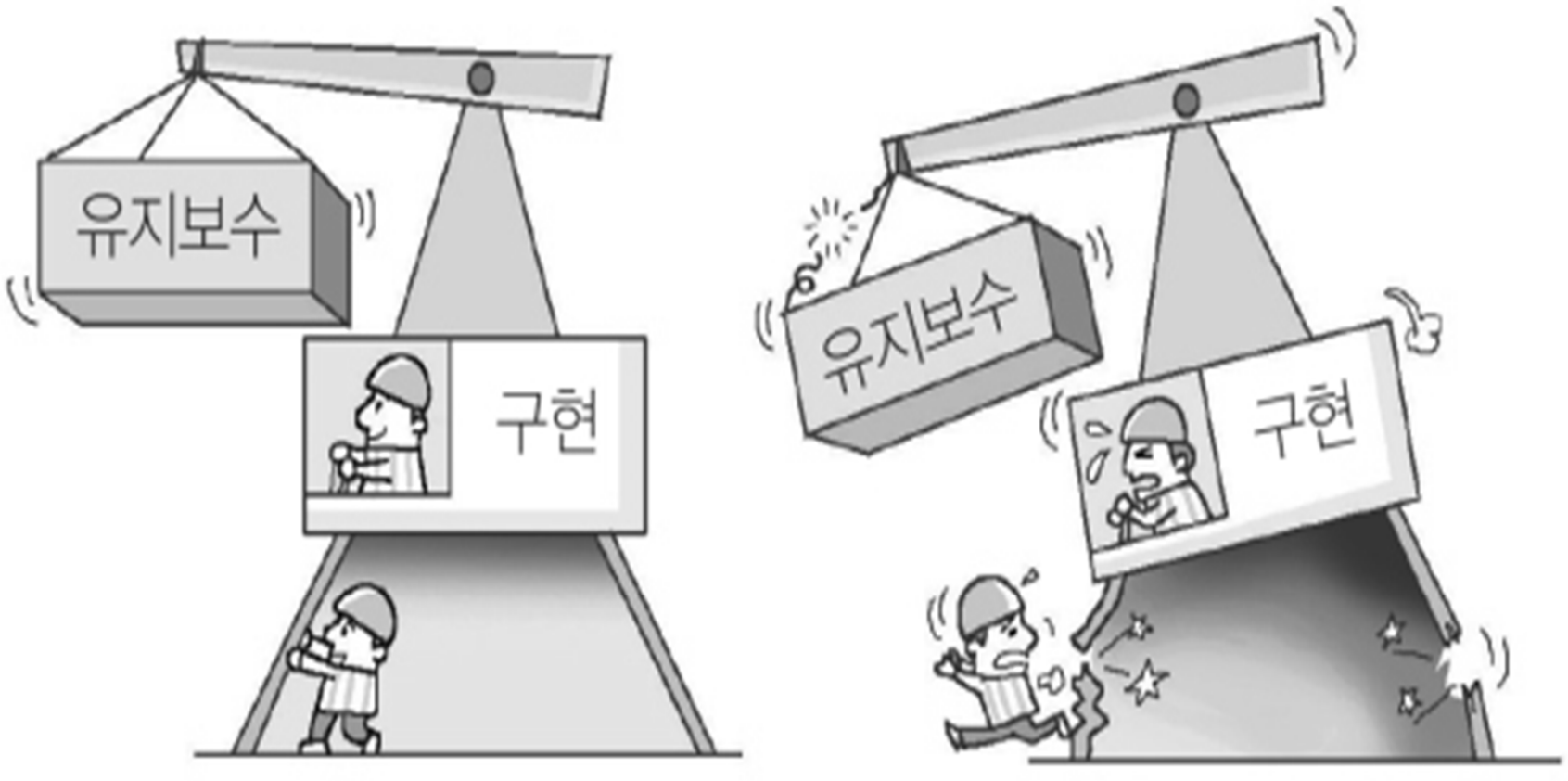
12장 학습 목표

- 설계의 관리적 측면 및 단계
- 설계의 기술적 측면 4가지 활동
- 설계 시 고려할 기본 원칙
- 응집도 정의 및 스펙트럼
- 결합도 정의 및 스펙트럼
- 기능적 독립성, 응집도, 결합도 간의 관계 설명

소프트웨어 설계 단계



설계의 중요성



(a) 설계가 잘 되어 있을 때

(b) 설계가 잘못되어 있을 때

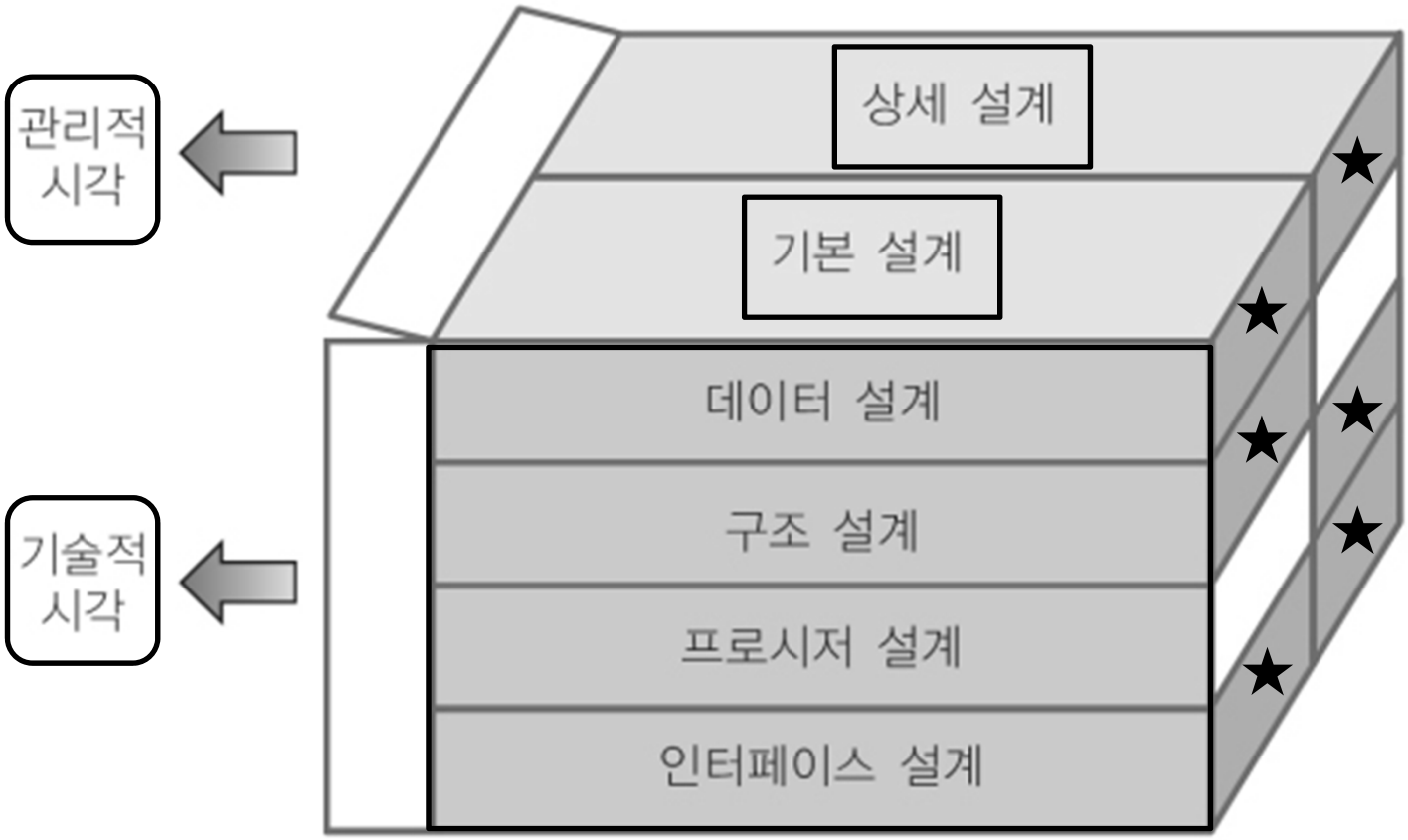
12.1 소프트웨어 설계 개요

- 시스템 설계(System Design)
  - 시스템 구현을 위해서 시스템을 여러 서브시스템(Subsystem)으로 나누고 서브시스템 요소를 하드웨어와 소프트웨어에 할당하는 것
  - 시스템 전체 구조를 먼저 결정하고, 다음은 구체적인 설계 진행
  - 소프트웨어에 할당된 서브시스템을 설계하는 과정
  - 요구사항 분석은 응용(업무) 분야의 개념에 초점, 소프트웨어 설계는 컴퓨터 개념으로 중심 이동
  - 분석 단계에서 밝혀진 요구사항은 설계를 위한 골격 제공, 설계는 그 위에 살을 붙여가는 과정
  - 요구사항 해결을 위해 최소한 한가지 해결 방안은 있어야 하며, 가능하다면 다양한 해결방법 모색 및 평가 필요
  - 소프트웨어 설계자는 여러 해결 방법 중 수행 시간, 기억장치 및 다른 비용과 자원들을 최소화 할 수 있는 방법 선택

12.1 소프트웨어 설계 개요

- 서브시스템(Subsystem)
  - 일반적으로 상위레벨에서 시스템을 분할한 시스템 구성요소
  - = 자료 + 제어구조
  - 독립적으로 기능을 수행할 수 있고 컴파일 될 수 있는 프로그램 구성요소
  - 어떤 서비스를 제공하는가에 의해 구별, 서비스는 공통적인 목표를 제공하기 위해 필요로 하는 기능들의 모임

소프트웨어 설계의 기술적 측면과 관리적 측면



12.2 소프트웨어 설계 활동

- 관리적인 관점(2단계)

① 1단계 : **기본 설계**(Preliminary Design) 단계

- 소프트웨어 시스템의 구조와 데이터를 규명하며 사용자 인터페이스 정의(기본 설계 문서)
- **상위 설계**(High Level Design)라고도 하며, 기본 설계에서 상세 설계로 진행하면서 시스템 추상화 수준 낮춤
- 기본 설계는 설계 과정이 진행될수록 명세서에 더 구체적인 내용이 추가되어 설계 과정의 최종 산출물은 시스템 구현의 기초

② 2단계 : **상세 설계**(Detail Design) 단계로써 모듈의 구체적인 알고리즘에 초점(상세 설계 문서)

12.2.1 설계의 기술적인 관점

- 기술적인 관점(4가지 활동) : 데이터 설계, 구조 설계, 프로시저 설계, 사용자 인터페이스 설계
  - 데이터 설계(Data Design) : 요구사항 분석 단계의 정보 모델링에서 밝혀진 정보를 이용하여 자료구조와 데이터베이스 설계
  - 구조 설계(Architectural Design) : 기능 모델링과 동적 모델링에 나타난 결과를 이용하여 프로그램 구조상에 있는 각 구성요소(모듈) 간의 관계 기술
  - 프로시저 설계(Procedural Design) : 각 모듈 내부가 구체적으로 밝혀지며 어떤 알고리즘을 사용할지 결정
  - 사용자 인터페이스 설계(User Interface Design) : 사용자가 시스템 기능에 접근할 수 있도록 하는 사용자 인터페이스 설계

설계 가이드라인

- 소프트웨어 구성요소(모듈로도 부름) 간에 효과적 제어를 가능하게 하는 계층 구조 가짐
- 논리적 분할되어 모듈화(modular)되며, 일반적으로 기능에 의한 모듈화가 이루어져 모듈 간 계층 구조 형성
- 모듈 간, 또는 외부 환경과의 인터페이스가 최소화 되도록 설계해야 하며, 이는 모듈 내부의 응집도는 높아야 하고, 모듈들 사이의 결합도와 인터페이스는 최소화될 수 있게 설계되어야 함
- 분석 과정에서 나타난 결과를 활용하여 설계가 이루어져야 하며, 설계는 요구사항 분석과정의 연장선상에서 봐야 하며, 요구사항 실현을 위해 분석 결과에 살을 붙여나가는 과정

12.2.2 설계의 관리적 관점

- 기본 설계
  - 설계에 대한 경험이 많은 엔지니어가 하는 것이 일반적
  - 기본 설계 이후 상세 설계는 각 개발자가 분담 수행
  - 우선 고려될 수 있는 것이 데이터에 대한 설계
  - 시스템에 필요한 정보(또는 객체)를 자료구조와 데이터베이스 설계에 반영
  - 사용자 인터페이스를 설계하는 것이 바람직함
  - 사용자 요구사항을 만족시킬 수 있도록 시스템 구조 설정

12.3 설계의 고려 사항

- 소프트웨어 설계에 사용되는 기본 원리들이 있으며, 소프트웨어 설계는 품질에 직접적인 영향을 미치며, 다음 단계인 구현, 시험 및 유지보수와 밀접한 관계
- 컴퓨터 엔지니어(프로그래머 포함)의 지혜는 작동하는 프로그램을 얻는 것과 올바르게 만드는 것의 차이점을 깨닫는 것에서 시작(Michael Jackson)
- 소프트웨어도 개발과 운용, 유지보수를 효과적으로 하기 위해 구성요소(모듈, 엔터티 등)들로 **분할**(Partition)
- 이들 요소들은 각기 서로 다른 기능을 수행하는 독립성(Independence)을 가져야 함
- 분할에서 추구하는 중요한 원칙

> "서로 연관되어 있는 부분들은 같은 구성요소에 있어야 하며, 서로 연관성이 없는 부분들은 연관성이 없는 구성요소들에 할당되어야 한다."

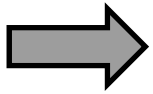
12.3.1 추상화(Abstraction)

- 엔지니어링 전 과정에서 이루어지는 중요한 원리
- 엔지니어링 : 추상화가 높은 단계에서 낮은 단계로 이동되는 과정
- 설계 과정에서도 높은 추상화 단계에서 낮은 추상화 단계로 진행하면서 단계적인 정제(Refinement)가 이루어짐
- 기본 설계에서 상세 설계로 진행되면 추상화 수준은 낮아지고, 원시코드가 생성될 때 추상화는 최하위 수준
- 설계 단계에서 주로 사용하는 추상화 : **제어(Control) 추상화, 절차(Procedural) 추상화, 데이터(Data) 추상화**

추상화의 예 : 제어 추상화

- 두 개 변수 값을 교환하는 제어 구조 및 제어구조 추상화

```
temp   := value1;
value1 := value2;
value2:= temp;
```



```
value1 ⇔ value2;
```

- 이런 추상화 메커니즘은 상세한 순서적 제어 구조의 내부 정보를 감추었지만 설계 단계에서의 명세 수준으로는 충분하며 이를 이용하여 다음 단계에서 쉽게 코딩 가능

예) 어떤 회사의 모든 사업부에 대해 필요한 처리를 하는

순환 제어 구조 추상화 예

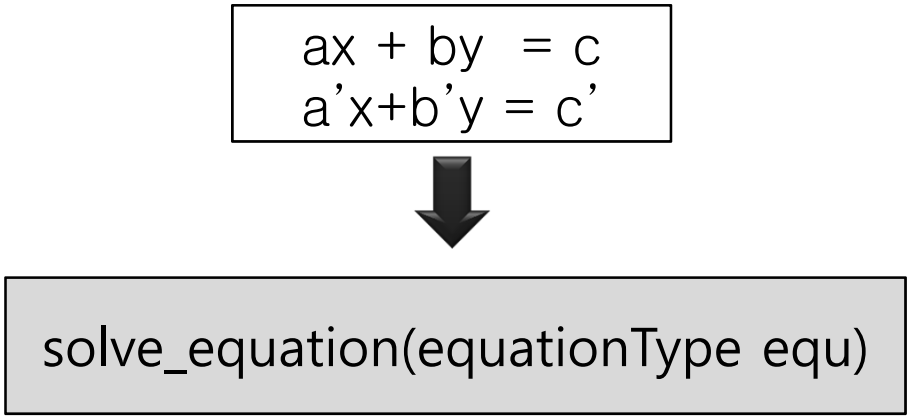
- 순환 제어구조를 추상화시키면 회사의 각 사업부를 어떻게 접근하는가에 대해서 언급하지 않아도 됨

```
for (회사의 모든 사업부) do
...
end for
```

추상화의 예 : 절차 추상화

- 절차 추상화 : 어떤 기능의 수행 과정을 추상화 하는 것

  예) 방정식의 해를 구하는 과정의 절차 추상화로 표시



추상화의 예 : 데이터 추상화

- 데이터 추상화 : 제어 추상화 또는 과정 추상화와 같이 상세 정보(데이터)를 감추는 것(프로그래밍 언어가 지원)

```
const int MAXSIZE = 100;

class stack
{
private:
      char stack_value[MAXSIZE];
      int top;
public:
      stack()top = 0;;
      void push(char);
      char pop();
};

void stack::push(char x)
{
  if (top+1 == MAXSIZE)
     error("stack is full");
  else
     stack_value[++top] = x;
}
```

```
char stack::pop()
{
  if (top == 0){
     error("stack is empty");
     return NULL;
  }
  else
     return stack_value[top--];
}

void main()
{
      stack st1;
      char x, y;

      st1.push('a');
      st1.push('b');
      x = st1.pop();
      y = st1.pop();
      cout << x << y << endl;
}
```

12.3.2 정보 은닉(Information Hiding)

- 필요하지 않은 정보는 접근할 수 없도록 해서 한 모듈 또는 하부 시스템이 다른 모듈의 구현에 영향을 받지 않게 설계되는 것
- 소프트웨어 설계 단계에서 채택되는 설계 전략(Design Strategy)을 지역화(Localize)하여 설계 전략에 변경이 발생하는 경우 최소한의 모듈에만 영향이 미치도록 하는 것
- 정보 은닉은 모듈들 사이의 독립성을 유지시켜 주며, 모듈 내부의 자료 구조나 수행 방법이 변경되더라도 그 모듈에서 제공하는 인터페이스(오퍼레이션)를 사용하는 외부 모듈은 영향을 받지 않도록 해줌
- 설계에서 은닉되어야 할 기본 정보
  - 상세한 데이터 구조
  - 하드웨어 디바이스를 제어하는 부분
  - 특정한 환경에 의존하는 부분(예를 들면 특수한 운영체제에 의존하는 부분 또는 특정한 DBMS에 의존하는 부분 등)
  - 물리적 코드(예를 들면 IP 주소, 문자코드 등)

정보 은닉을 이용한 스택 예 : 프로그래밍 언어에서 지원(C++, JAVA)

```
#define MAXSIZE  100;
struct STACK{
      char stack_value[MAXSIZE];
      int top;
};
typedef struct STACK stack;

void create_stack(stack aStack)
{
  aStack = new STACK;
  aStack->top = 0;
}
void push(stack aStack, char x)
{
  if (aStack->top+1 == MAXSIZE)
    error("stack is full");
  else    aStack->stack_value[++aStack->top] = x;
}
char pop(stack aStack)
{
  if (aStack->top == 0){
    error("stack is empty");
    return NULL;
  }
  else    return aStack->stack_value[aStack->top--];
}
```

```
char top_element(stack aStack)
{
      if (aStack->top == 0){
         error("stack is empty");
         return NULL;
      }
      else
         return aStack->stack_value[aStack->top - 1];
}

void destroy_stack(stack aStack)
{
      delete aStack;
}

void main()
{
      stack st1;
      char x, y;

      create_stack(st1);
      push(st1, 'a');
      push(st1, 'b');

      x = pop(st1);
      y = top_element(st1);

      destroy_stack(st1);
      printf("%c , %c", x, y);
}
```

정보 은닉이 정확히 되지 않은 스택 : 프로그래밍 언어에서 미지원(C)

```
#define MAXSIZE  100;
typedef struct {
      char stack_value[MAXSIZE];
      int top;
} stack;
void push(stack* aStack, char x)
{
  if (aStack->top+1 == MAXSIZE)
    error("stack is full");
  else
    aStack->stack_value[++aStack->top]   =
     x;
}
char pop(stack* aStack)
{
  if (aStack->top == 0){
    error("stack is empty");
    return NULL;
  }
  else
    return         aStack->stack_value[aStack-
     >top--];
}
```

```
void main()
{
      stack* st1;
      char x, y;

      st1 = new stack;
      st1->top = 0;
      push(st1, 'a');
      push(st1, 'b');

      x = pop(st1);
      y = st1->stack_value[st1->top - 1];

      delete st1;
      printf("%c , %c", x, y);
}
```

정보 은닉

- 시스템 설계에 있어서 구성요소 간의 독립성 유지한다는 점에서 중요
- 모듈 서로 간의 내부 구조를 감추어 주고(추상화), 서로의 내부 구조를 알 필요가 없이 오직 정해진 인터페이스로만 서로 소통
- 만약 어떤 모듈에 수정이 요구되는 경우 모듈 내부의 자료구조와 이에 접근하는 동작들에만 수정을 국한을 시킴으로써, 변화에 쉽게 적응할 수 있고 유지보수를 용이하게 해나갈 수 있게 하는 기반 제공
- 일반적으로 소프트웨어 설계할 때 계층구조를 이용하여 설계하는 것도 계층들 사이의 정보 은닉을 얻기 위함
- 결국 모듈 내부의 자료구조나 수행방법 등에 변화가 일어 났을 때 외부 모듈들이 영향을 받지 않도록 설계할 수 있도록 지원하는 개념
- 만약 모듈의 논리적인 수정(예: 요구사항 변경)이 요구되는 경우는 어떻게 될까?
- **캡슐화**(Encapsulation) : 객체 지향 개발방법의 중요한 개념 중의 하나로써 정보 은닉을 통한 추상화, 독립성 향상을 얻을 수 있는 방법

12.3.3 단계적 정제(Stepwise Refinement)

- 하향식 설계방법에 주로 사용
- 기본 설계 단계에서 나타나는 프로그램 구조에서 점차 모듈에 대한 세부 사항으로 내려가며 구체화
- 정제 과정에서 추상화 수준은 낮아지며 각 기능은 분해되어 해결방안 제시
- 많은 노력을 들이는 과정이며, 세부적인 묘사를 가능하게 함으로써 시스템 구현을 가능케 함
- 기본 설계나 상세 설계는 높은 추상화 단계에서 낮은 추상화 단계로 가는 단계적 정제 과정
- 문제기술에서 요구사항 분석, 설계, 프로그래밍으로 이어지는 엔지니어링의 흐름도는 단계적 정제 과정
- 레벨화(Leveling)/계층화 : 일반적으로 큰 시스템을 상세화하면서 계층적인 배열을 두어 상호 종속관계를 표시하는 것

12.3.4 모듈화(Modularity)

- 모든 공학 분야에서 대부분의 시스템을 구성요소로 나누어서 접근
- 소프트웨어 구성요소를 대표하는 것은 **모듈**(Module)
- 소프트웨어 모듈 : 프로그래밍 언어로 표현할 때 서브루틴(Subroutine), 프로시저(Procedure), 함수(Function) 등으로 불림
- 시스템을 모듈화할 때 하향식 접근방법을 사용하여 기능 단위로 쪼개어 나가는 것이 일반적
- 시스템을 지능적(Intelligently)으로 관리할 수 있도록 해주며, 복잡도(Complexity) 문제 해결

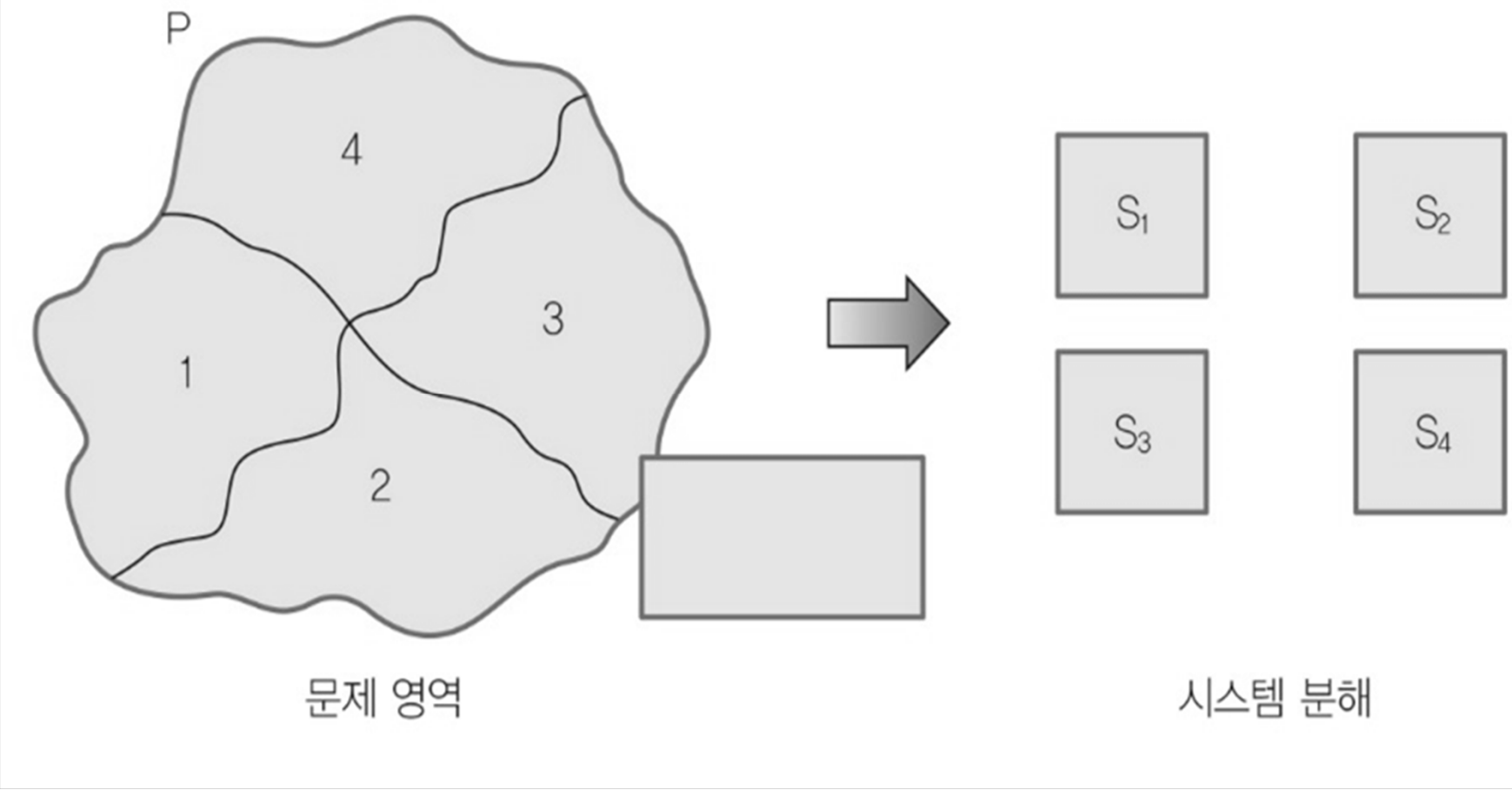
12.3.4 모듈화(Modularity) (계속)

- 크고 복잡한 문제를 해결하기 위해 문제를 작은 단위인 모듈로 분할하여 정복(Divide and Conquer)
- 시스템 유지 보수와 수정 용이
- 모듈 수가 증가하면 상대적으로 각 모듈 크기는 감소하며 모듈 사이의 상호교류가 증가하여 시스템 성능이 떨어지고 과부하(Overload) 현상 발생
- 시스템 특성을 파악하여 기존 시스템들의 경험과 가이드라인 활용
- 기존 시스템들을 살펴보면, 시스템 특성에 따라 몇 가지의 구조적인 틀이 있고, 이 틀들을 이용하면 유사한 특성을 가진 시스템을 만들 때 노력과 시간 절약

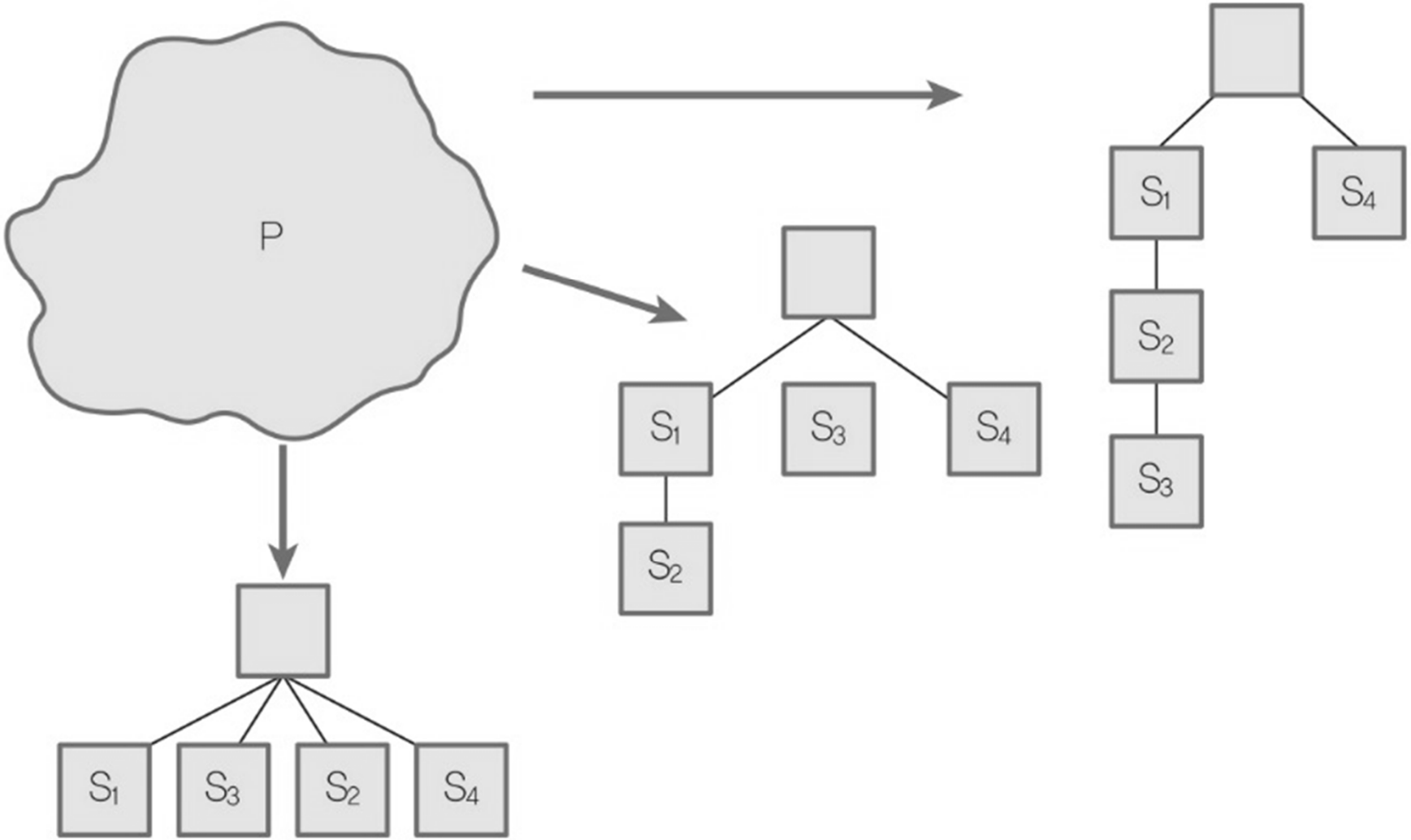
12.3.5 프로그램 구조화(Program Structure)

- 분할 과정에 의해 얻어질 수 있으며, 분할 정복 개념과 연관
- 분할과정은 앞의 요구사항 분석 과정에서 일차로 이루어졌으며, 설계 단계에서 더욱 세분화
- 시스템 중요 요소나 기능을 찾아내어 분할해 나가는 것은 분석가 임무
- 분석 결과를 구조화시키는 것은 설계자 임무
- 시스템을 어떻게 분할해 나갈 것인가는 간단한 문제가 아니며 어떻게 분할하면 좋은지 완벽한 가이드라인은 없음

문제 영역 분해



문제에 대한 다양한 프로그램 구조화

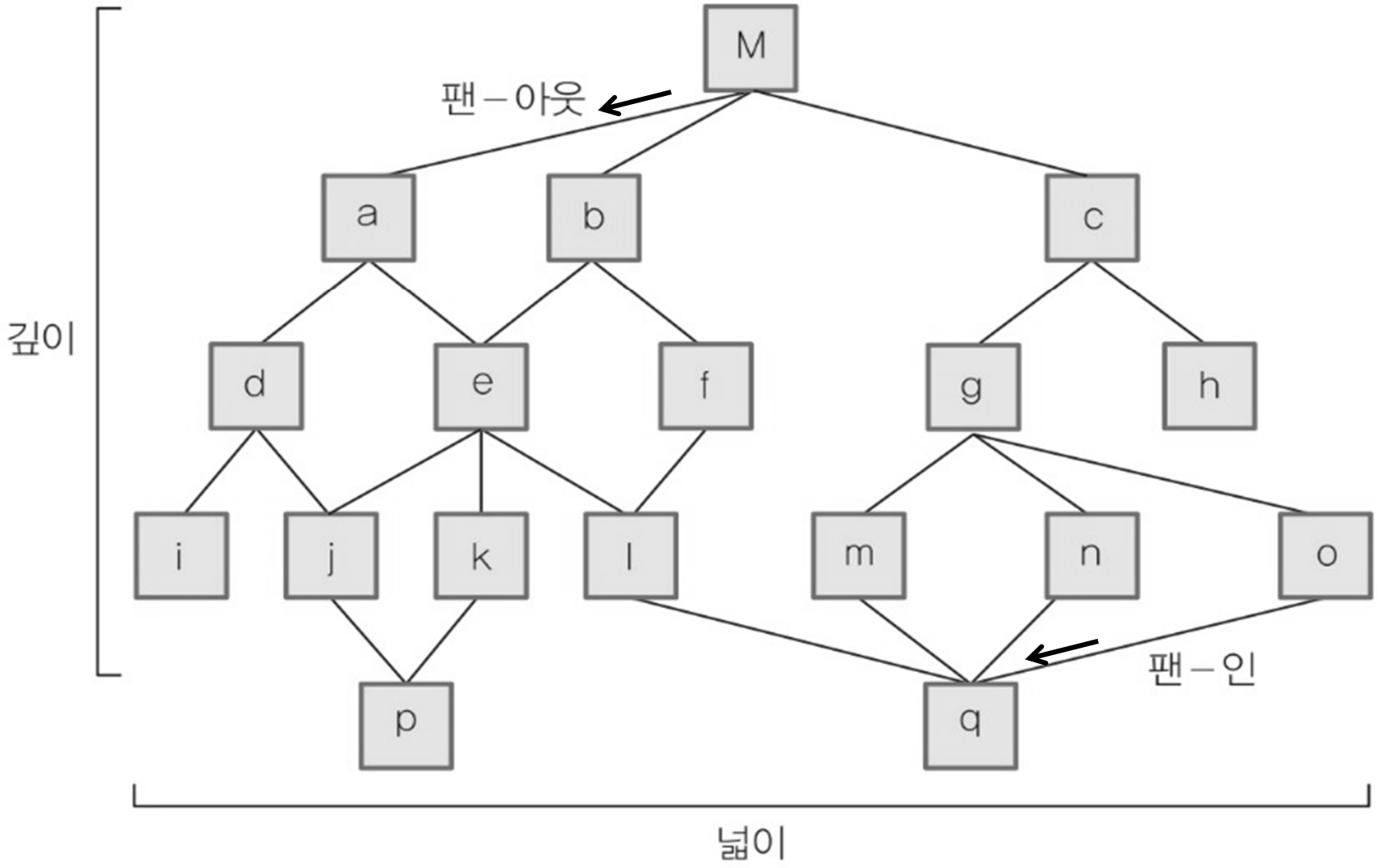


프로그램 구조에 사용하는 용어

- 제어 계층을 가진 프로그램 구조는 프로그램 요소인 모듈들 사이의 계층적 체계를 나타내주며 트리(Tree)와 유사한 그림으로 표시
- 사각형 : 모듈, 선(–) : 제어 관계
- 프로그램 구조 측정 및 표현 용어

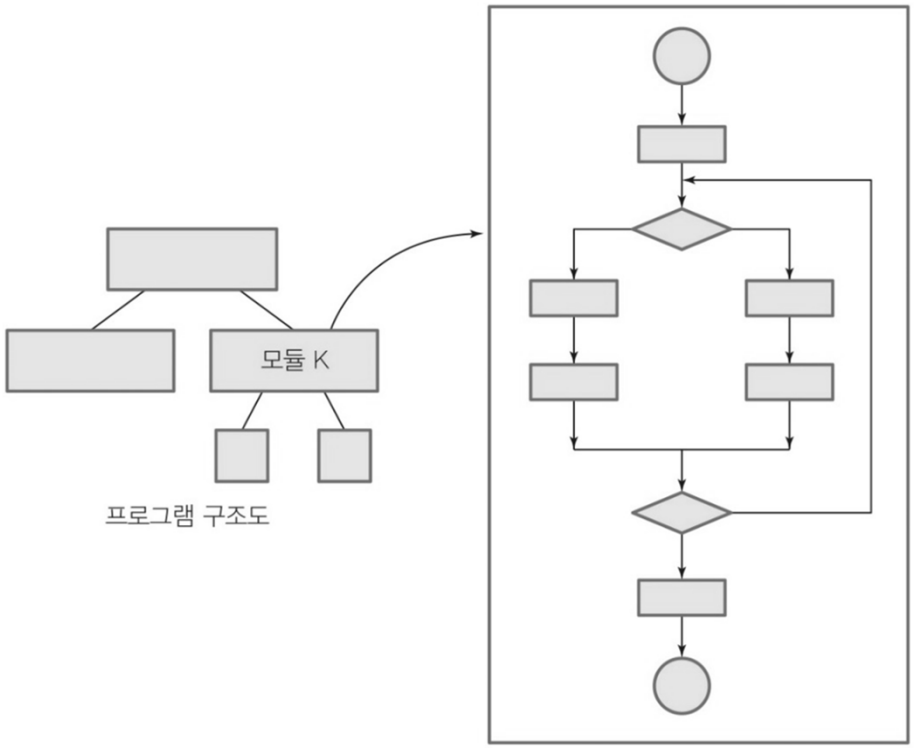
깊이 (depth)	제어 단계의 수
넓이 (width)	제어의 전체적인 폭
팬-아웃 (fan-out)	한 모듈이 직접 불러 제어하는 하위 계층 모듈수
팬-인 (fan-in)	주어진 모듈을 직접 불러 제어하는 상위 조정 모듈수

프로그램 구조 예 : 관련 용어



구조도와 순서도

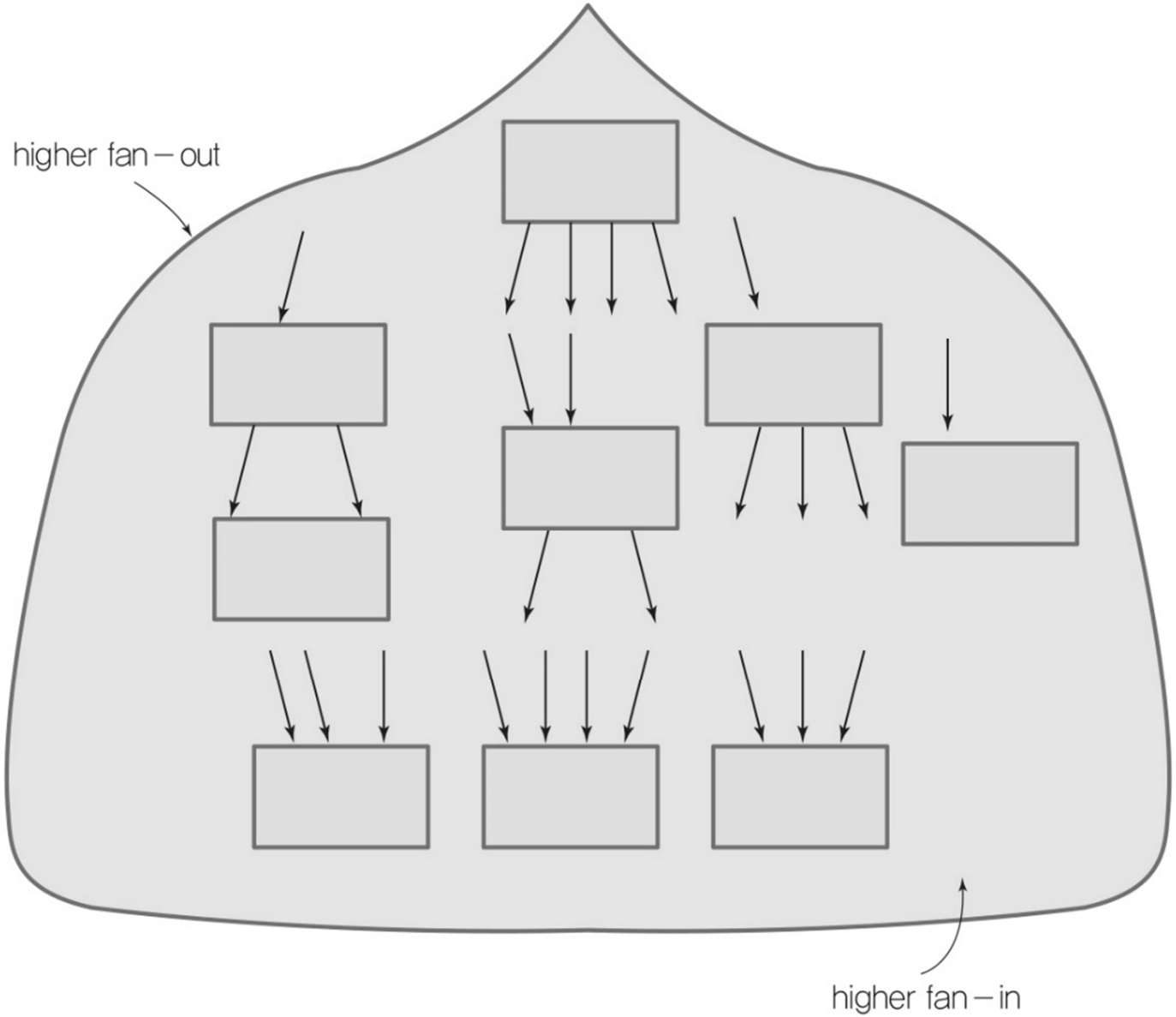
- 구조도 : 모듈(프로그램) 구조를 나타내는데 사용
- 순서도 : 프로그램 구조가 만들어진 후 각 모듈들에 대한 구체적 절차 규명, 모듈 내부의 작업절차 나타내는데 이용



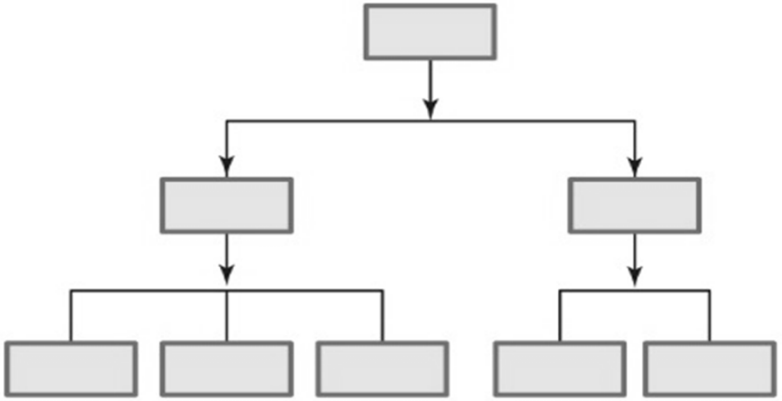
프로그램 구조도

모듈 K 순서도

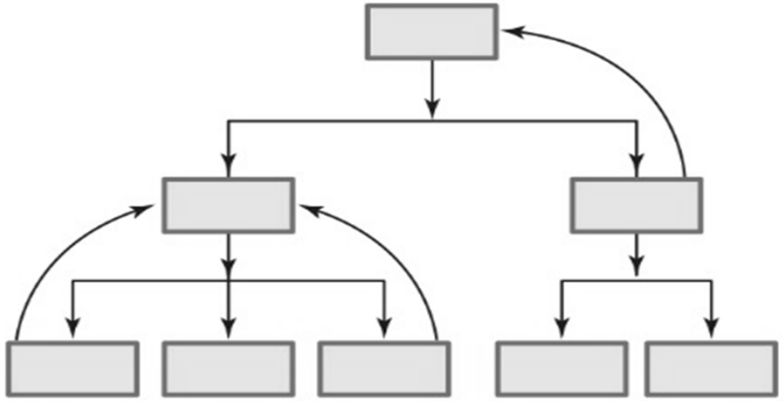
모스크 시스템 구조



데이터베이스 설계 : 목표 데이터베이스 적용



(a) 계층 구조

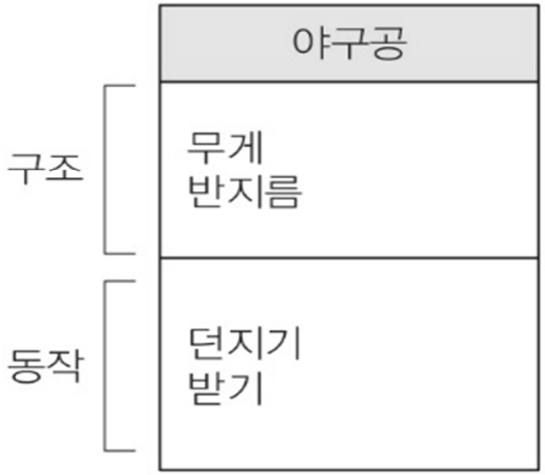


(b) 네트워크 구조

학생

이름	주민등록번호	생년월일	주소

(c) 관계형 구조



(d) 객체지향 구조

12.4 설계의 품질 요소

- 좋은 설계
  - 효율적으로 프로그램을 할 수 있게 하여 주는 설계
  - 소프트웨어의 진화 문제를 잘 해결할 수 있도록 변화에 쉽게 적응할 수 있는 설계
  - 설계 결과인 설계 문서는 읽기 쉽고 이해하기 쉽게 작성되어야 하며, 시스템 변화 시 영향은 국소화(Local)
  - 모듈들은 서로 독립적이며, 구성요소 내부 응집력 최대화
  - 모듈들 사이의 연결을 나타내는 결합도 최소화
  - 모듈들 사이의 결합도가 약할수록 모듈의 독립성 증가, 새로운 환경에 적응할 수 있는 적응력 증가
  - 기능적 독립성, 응집도 극대화하고, 모듈들 사이 결합도 줄이는 것이 유지보수를 쉽게 할 수 있게 하는 우수한 설계 원칙

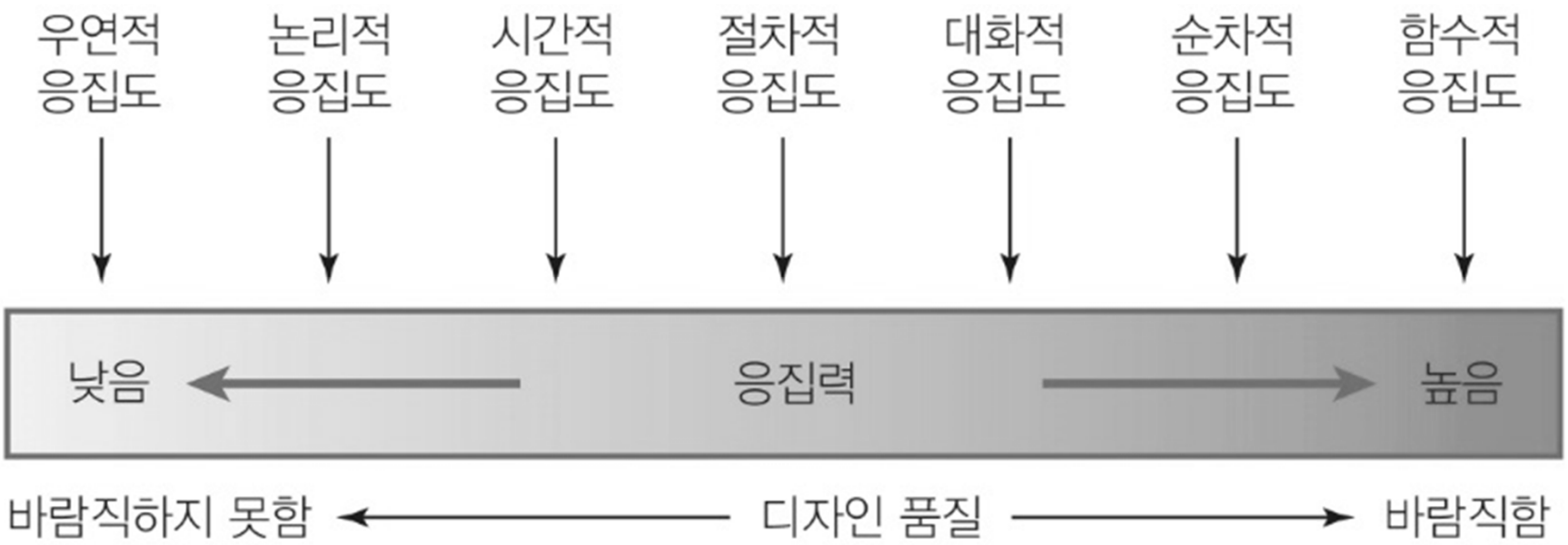
12.4.1 모듈 독립성(Module Independence)

- 소프트웨어 구성요소가 독립성을 가지고 기능 수행 시 성숙된 모습이 나타남
- 소프트웨어 구성요소의 기능적 독립성은 모듈화 과정과 정보은닉 개념에서 나타나는 부산물
- 모듈이 1개의 기능만 수행하며 다른 모듈과 상호 교류와 결합을 최소화시킬 때 모듈의 기능적 독립성 극대화
- 모듈 간 관계가 많고 복잡하면,
  - 소프트웨어 설계가 복잡하여 설계에 투여되는 노력이 많이 들고 설계 비용 증가
  - 소프트웨어 이해가 어렵고 작은 설계 변경에도 많은 모듈에 영향을 주어 유지보수 어려움
- 모듈 간 관계가 최소화되고 단순한 경우 모듈 독립성 높음
- 설계 목표는 이해하기 쉽고 수정이 쉬운 소프트웨어를 만드는 것이며 높은 독립성을 갖고 있는 모듈들을 설계하는 것은 기본

12.4.2 설계 응집도(Cohesion)

- 모듈 내부가 얼마나 단단히 뭉쳐져 있는지 나타내는 성숙도의 측정치
- 모듈 안의 구성요소들이 어울리는 정도로써 구성요소를 묶어주는 **시멘트**(Cement)
- 모듈이 하나의 임무를 수행하는 정도를 나타내는 것으로 모듈의 독립성을 측정하는 또 다른 척도
- 모듈 내부 요소 간 응집도 증가하게 설계
- 모듈이나 시스템 구성요소는 1개의 논리적 기능 수행 및 하나의 논리적 엔티티를 나타내는 것이 바람직
- 모듈 응집도를 높이면 모듈 간 낮은 결합도를 얻을 수 있으며, 낮은 응집도는 높은 결합도 발생
- 소프트웨어 설계 시 모듈들은 높은 응집도 갖고, 모듈들 사이의 결합도가 낮게 하는 것이 바람직

응집도 스펙트럼


우연적 응집도
논리적 응집도
시간적 응집도
절차적 응집도
대화적 응집도
순차적 응집도
함수적 응집도
낮음
응집력
높음
바람직하지 못함
디자인 품질
바람직함

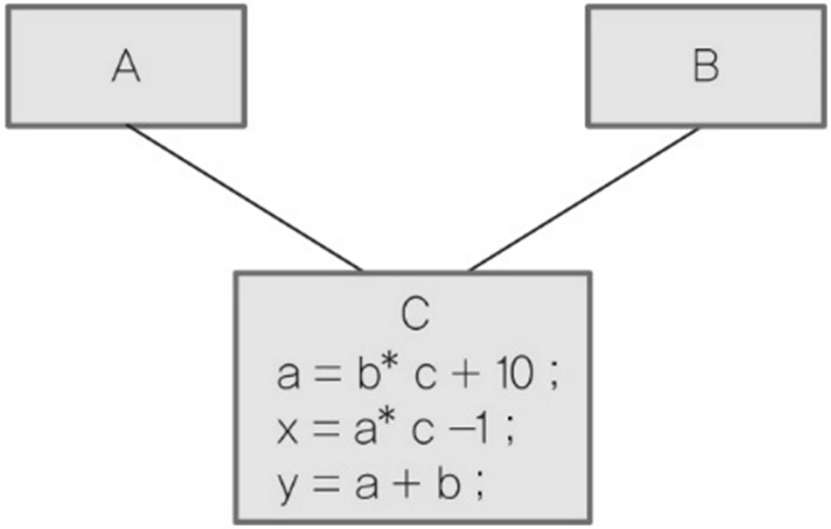
우연적 응집도(Coincidental Cohesion)

- 모듈 내의 구성요소가 뚜렷한 연결성을 가지고 있지 않음
- 극단적인 예로써 임의로 20줄씩 잘라 모듈을 만들었을 때 우연적인 응집도 밖에 기대할 수 없음
- 뚜렷한 의미나 기능이 없지만 여러 모듈에 유사한 명령문들이 중복되어 있는 경우에 단지 중복되는 부분을 줄이려고 이를 묶어 모듈을 설계하였다면 이런 모듈은 우연적 응집도 갖음

우연적 응집도 (계속)

예) 모듈 *A, B*의 유사 명령문들을 모아 모듈 *C* 구성한 예

- 이런 설계는 모듈 *A, B*에 대한 수정이 발생하지 않는다는 틀린 가정에서 출발
- *A*나 *B*의 알고리즘 변경이 발생 시 *C* 모듈의 수정이 불가피하며, *C*를 수정하는 것이 매우 어려움
- 설계에서 우연적 응집도는 반드시 피해야 함



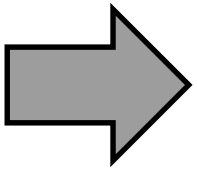
논리적 응집도(Logical Cohesion)

- 설계 시 논리적으로 연관된 임무나 비슷한 기능(예: 입출력, 오류처리 등 비슷한 기능들)을 묶어 한 모듈 구성
- 만약 여러 기능 중 어느 한 기능에 변경이 발생하였을 때 모듈 수정 필요
  - 예) 3개의 계산식 모두 6x + 4라는 식을 포함하고 있기 때문에 1개의 모듈로 이 방정식을 푸는 임무 수행

$y = 5x^2 + 6x + 4$

$y = 6x^3 + 6x + 4$

$y = 7x^4 + 6x + 4$



```
long solve_equation(int no_equ, long x)
{
        long y;
        y = 0;
        switch (no_equ)
                case 1:  y = 5 * x * x;
                         break;
                case 2:  y = 6 * x * x * x;
                         break;
                case 3:  y = 7 * x * x * x * x;
                         break;
        }
        y = y + 6 * x + 4;
        return y;
}
```

시간적 응집도(Temporal Cohesion)

- 모듈 내 구성요소들이 서로 다른 기능을 같은 시간대에 함께 실행하는 경우

  예) 초기화 모듈은 흔히 볼 수 있는 시간적 응집도 모듈, 초기화 모듈 *Init_Variables*는 변수에 대한 초기화 작업을 수행하는 시간적 응집 모듈이며, 각 변수들에 대한 초기화는 서로 연관성이 없음

- 시간적 응집도를 가진 모듈도 여러 가지 임무를 수행하지만 같은 시간대에 실행한다는 점에서 이런 응집도는 우연적 응집도 또는 논리적 응집도 모듈 보다 응집도가 높음

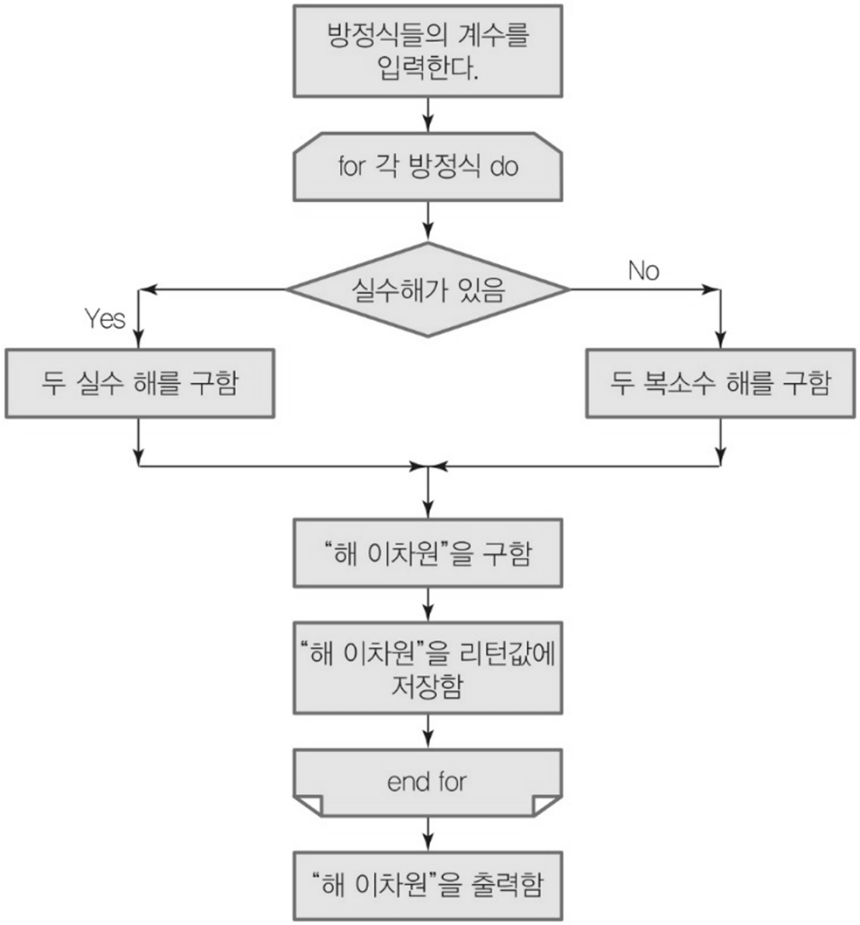
```
void Init_Variables()
{
    ...
    no_student      = 0;
    no_department = 0;
    university_name = "Chungnam National University";
    ...
}
```

절차적 응집도(Procedural Cohesion)

- 모듈 내 구성요소 간 연관성이 있고, 특정 순서에 의해 수행되어야 하는 경우
- 프로세스의 절차적인 요구조건에 따라 모듈을 설계할 경우 모듈이 단지 절차의 한 부분이라는 이유로 설계가 되었다면 응집도가 낮은 절차적 응집도를 가진 모듈임

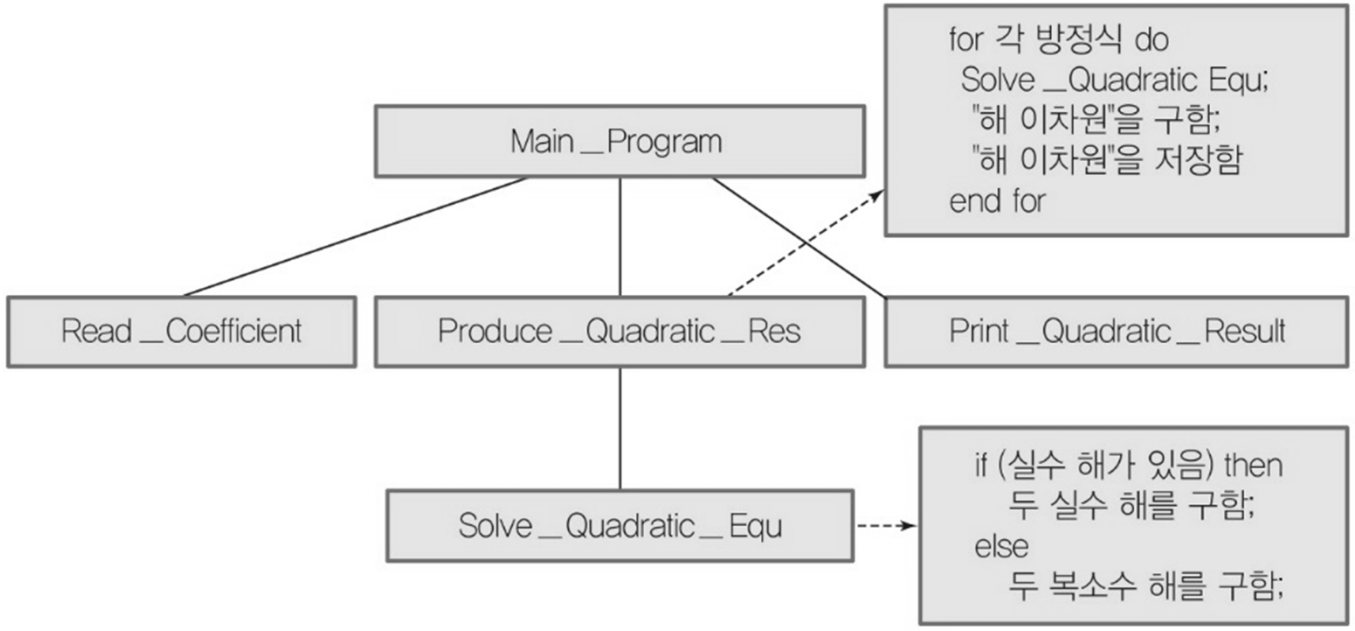
예) 방정식들의 계수를 입력하여 이차 방정식들을 풀고(근: $a+b_i$, $a-b_i$) 해

이차원 $a^2 + b^2$을 구하는 경우



절차적 응집도 : Solve_Quadratic_Equ 모듈 예

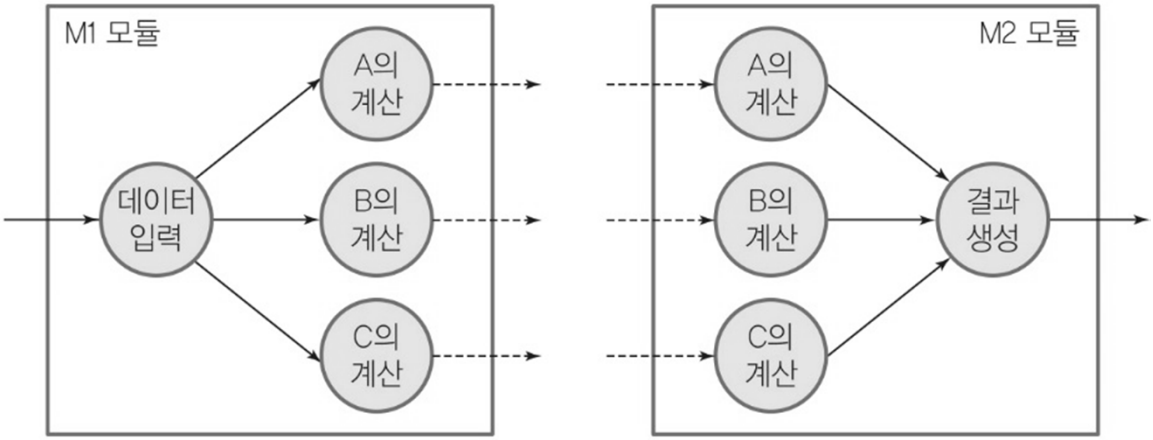
- 이 모듈의 구성요소는 순환구조의 한 원소이기 때문에 절차적 응집 모듈 특성
- 프로세스 절차에 의해서만 모듈화를 하게 되면 설계된 모듈이 여러 임무를 수행할 수 있거나 한 임무의 일부분 수행
- 예에서 2차 방정식을 푸는 임무는 "해 2차원"을 구하는 임무의 일부분이지만 데이터는 고려하지 않고 프로세스 절차만 고려하여 *Solve_Quadratic_Equ*는 "해 2차원"을 구하는 임무 전부를 수행하지 않음



대화적 응집도(Communicational Cohesion)

- 모듈이 여러 가지 기능을 수행하며 모듈 내 구성요소들이 같은 입력자료를 이용하거나 동일 출력 데이터를 만들어 내는 경우
- 절차적 응집도는 오직 프로세스 절차만 고려하여 얻은 모듈이지만 대화적 응집 모듈은 모듈화에서 프로세스 절차와 데이터를 동시에 고려하여 모듈화된 것으로써 해결하려는 문제와 많은 연관성을 갖고 있어 절차적 응집도보다 높은 응집도를 갖는 것으로 볼 수 있음

예) DFD로 표시된 정형적인 대화적 응집 모듈로써 모듈 *M1*은 같은 입력을 갖고 있고 모듈 *M2*는 같은 출력



대화적 응집도 (계속)

예) 행렬의 값을 입력하고 전치행렬 A^T와 역행열 A^{-1}을 계산하는 *Compute_Matrix*

```
void Compute_Matrix(long transform_matrix[][], long inverse_matrix[][])
{
        long aMatrix[5][5];

        for (int i = 0; i < 5; i++)
          for (int j = 0; j < 5; j++)
                Read an element to aMatrix[i][j];
        transform_matrix =  aMatrix^T 를 계산한다;
        inverse_matrix =  aMatrix^-1를 계산한다;
}
```

함수적 응집도(Functional Cohesion)

- 한 기능을 수행하기 위해 각 구성요소들이 필요한 경우
- 이런 모듈 내부요소들은 가장 밀접히 연관되어 있고, 또한 높은 응집도 갖음. 이런 모듈은 오직 하나의 기능을 수행함으로써 이해하기가 쉽고 수정이 쉬움

예) "행열의 값을 입력하여 그의 역행열을 계산하여 출력하는" 프로그램을 "입력 모듈", "역행열 계산 모듈" 및 "출력 모듈"로 나누어 설계하면 각 모듈은 함수적 응집도를 갖게 되며, 다음은 평방근을 구하는 모듈로 함수적 응집도의 예

```
real procedure squareRoot(real)
{
    .....
    Calculate square root of the number given.
    .....
    return result;
}
```

응집도 요약

- 한 모듈 내에 여러 기능을 합쳐 놓으면 오류를 일으킬 가능성이 증가하며, 시험하기도 어렵게 됨
- 이것은 앞의 기능이 변경되었을 때 뒤의 기능에 영향을 주게 되며, 오류가 확산될 가능성이 커지기 때문
- 높은 응집도는 모듈의 기능적 독립성을 높여주며 소프트웨어의 수정과 확장이 용이해져 고품질의 소프트웨어를 만드는 기준이 됨
- 응집도는 모듈을 중심으로 분류, 설명되었으나 시스템에 사용되는 객체(또는 데이터)를 중심으로 응집도가 분류 가능
- 객체지향 개발방법은 객체의 정적인 데이터와 동작을 함께 묶어 객체를 정의함으로써 높은 응집도를 얻을 수 있는 개발 방법임(2학기 강의)

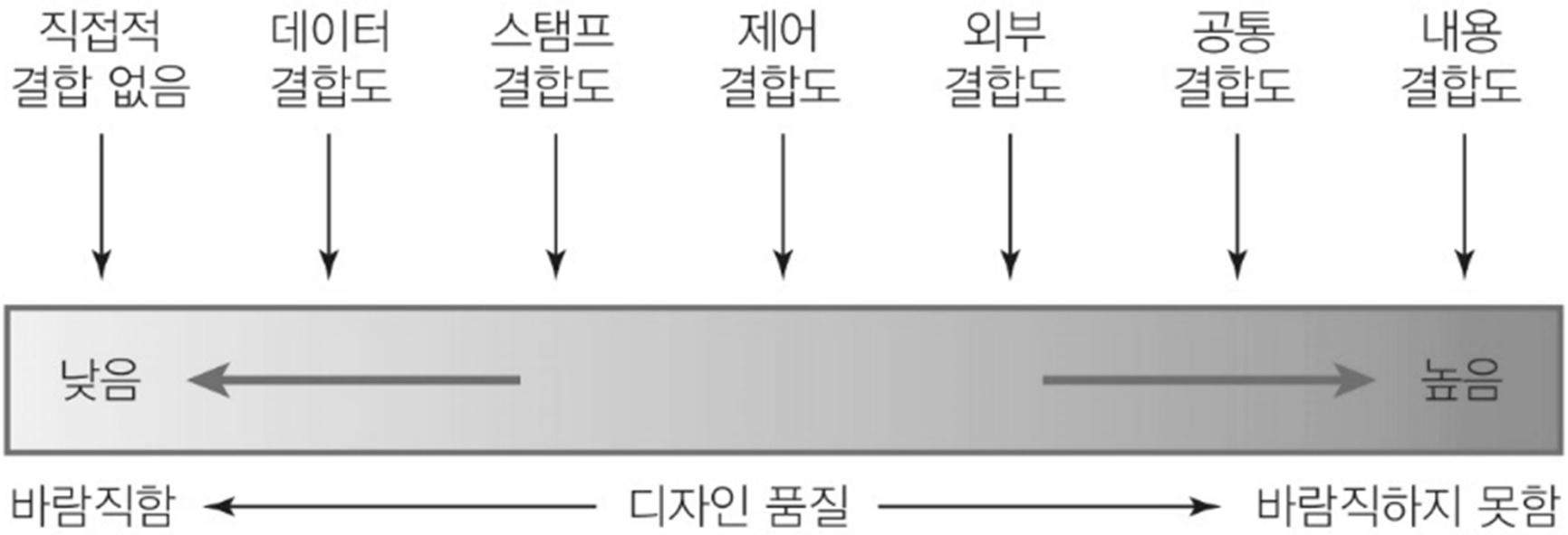
12.4.3 결합도(Coupling)

- 모듈 사이의 상호 연관성의 복잡도
- 모듈들 사이의 상호 교류가 많고 서로의 의존이 많을수록 모듈들 사이의 결합도는 높아지게 됨
- 특히 인터페이스가 정확히 설정되어 있지 않거나 기능이 정확히 나누어져 있지 않을 때 불필요한 인터페이스가 나타나 모듈 사이의 의존도는 높아지고 결합도 증가
- 시스템 구성요소 간의 관계도 각 구성요소의 독립성과 응집도를 바탕으로 구성요소 간의 관계가 맺어져야 함
- 결합도는 프로그램 요소들 사이의 상호 연관성을 표시하여 주는 방법이며, 모듈의 독립성 및 응집도와 밀접한 관계
- 만약 두 모듈이 서로 옆에 있건 없건 완벽하게 기능을 수행하는 경우라면 이들은 서로 완전히 독립적이라 할 수 있으며 이는 서로 상호 교류가 전혀 없음을 의미
- 결합도가 높을수록 한 모듈의 변화가 다른 모듈에도 영향을 주어 **파문 효과**(Ripple Effect) 일으킴
- 파문 효과가 클수록 시스템 유지보수가 어려워짐

결합도에 영향을 미치는 4가지 요소

- 모듈들 사이의 연결 유형
  - 모듈들 사이의 연결은 한 모듈 안에 있는 요소가 다른 모듈에 의해 참조될 때 발생
- 인터페이스의 복잡도
  - 각 인터페이스는 모듈들의 연결을 위해 꼭 필요한 정보만을 표시하여 복잡도가 최소화될 수 있도록 만들어져야 함
- 정보 흐름의 유형
  - 모듈들 사이에 교류되는 정보의 유형에는 크게 데이터와 제어 신호가 있음
- 바인딩 시간
  - 모듈 사이의 연결을 묶는 때를 바인딩 시간(binding time)이라 하며, 컴파일(compiling)할 때, 적재(loading)할 때, 실행(execution)할 때 등이 있으며, 실행 시간에 바인딩되면 로딩 시간에 바인딩된 것보다 결합도가 약하고, 적재시간에 바인딩되면 컴파일 시간에 바인딩된 것 보다 결합도가 약함

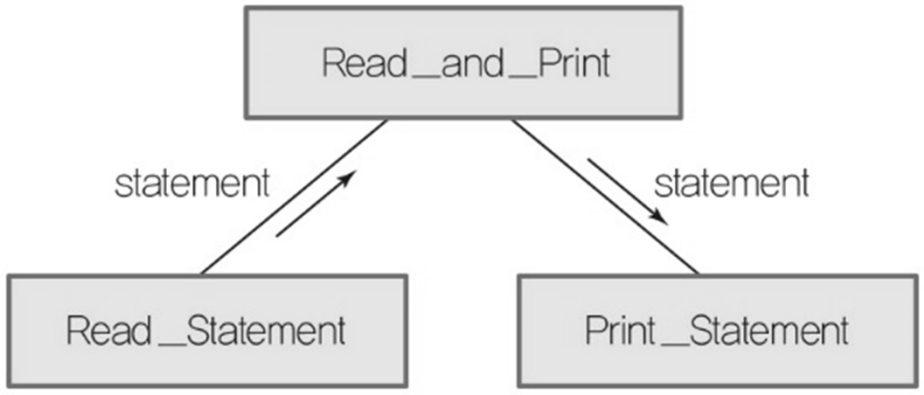
결합도 스펙트럼(Coupling Spectrum)



직접적 결합이 없음(no direct coupling)

- 서로 독립적이고 상호 교류가 없는 경우이거나, 두 모듈이 다른 모듈에 속해있어 직접적으로 연결되어 있지 않을 때

예) 모듈 *Read_char*와 *Print_char*는 *Read_and_Print* 모듈에 속하며 직접적인 결합이 없음



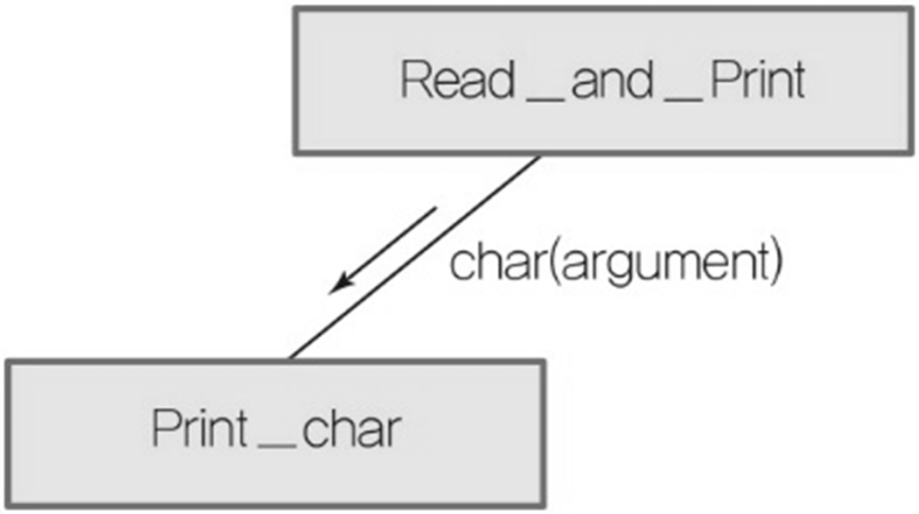

```
char Read_char()
    char ch;
    ch = 파일, 데이터 구조 혹은 단말기에서
    문자를 읽어 냄;
    return ch;
```

```
Print_char(char ch)
    …
    putchar(char);
    …
```

데이터 결합도(Data Coupling)

- 한 모듈이 간단한 데이터들을 매개 변수를 통해 다른 모듈과 주고받는 경우

예) *Read_and_Print* 모듈과 *Print_char* 모듈은 데이터 결합

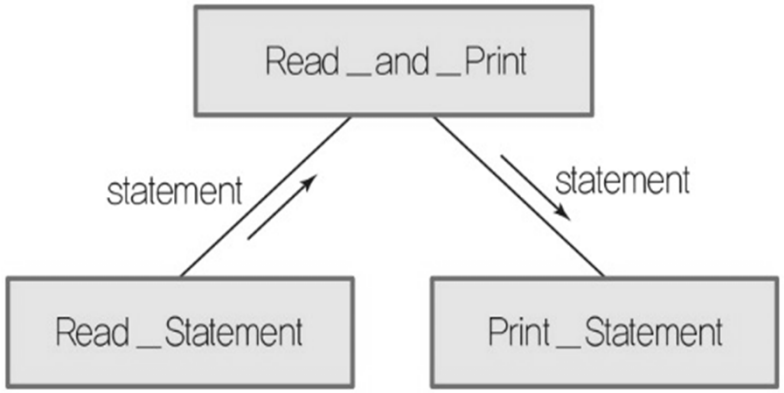



```
Read_and_Print()
    .....
    Print_char(char);
    .....
```

```
Print_char(char ch)
    .....
    .....
```

스탬프 결합도(Stamp Coupling)

- 스탬프 결합도는 레코드 또는 배열과 같은 복잡한 데이터 구조를 모듈 인터페이스를 통해 주고받는 경우
- 스탬프 결합도에서 호출 모듈과 피호출 모듈 간에 데이터 교환에서 사용되는 자료구조를 공통으로 쓰고 있기에 이런 자료구조에 대한 정보를 알아야 함

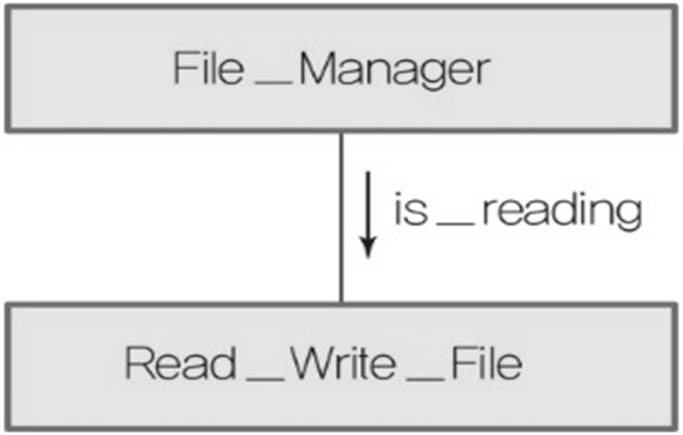



```
Read_and_Print()
{
  char aStatement[256];
   …
   Read_Statement(aStatement);
   …
   Print_Statement(aStatement);
}

void Read_Statement(char*aStatement)
{
      …
      Read a statement to the array aStatement;
      …
}
void Print_Statement(char* aStatement)
{
      .…
}
```

제어 결합도(Control Coupling)

- 모듈 사이에 제어 신호를 주고받아 다른 모듈의 행위를 변경할 수 있는 경우
- 제어 신호(Flag)가 하부 모듈에 전달되어 그 행위를 결정
- 데이터를 교류하는 것보다 제어 신호를 교류하는 경우 모듈들 사이의 결합도가 높음




```
void File_Manager()
{
        BOOL is_reading;
        ....
        is_reading = TRUE;
        Read_Write_File(filename, is_reading);
        ....
}

void Read_Write_File(char* filename, BOOL is_reading)
{
        ...
        if (is_reading){
                ...
                reading data from the file;
                ...
        }
        else{
                ...
                writing data to the file;
                ...
        }
        ....
}
```

외부 결합도(External Coupling)

- 입출력의 경우 모듈이 특수한 하드웨어에 결합되어 있거나, 통신 프로토콜, 운영체제(OS), 컴파일러 등과 같은 소프트웨어 이외의 다른 시스템 구성요소와 결합되어 있는 경우

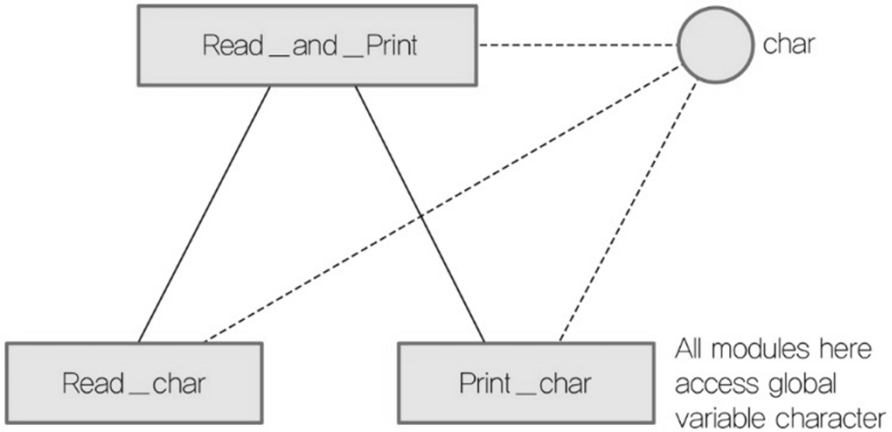
  예) 모듈이 컴파일러의 비표준 선택사양에 의해 컴파일 되는 경우(컴파일러 결합)나 표준화 되어 있지 않은 O/S 기능을 사용하는 모듈의 경우(예: 문서화되지 않은 MS-DOS 기능)가 이에 해당

공통 결합도(Common Coupling)

- 모듈들이 간단한 타입을 갖고 있는 전역 변수(Global Variable)를 사용 시

  예) Fortran의 COMMON 블록도 이에 해당

- 공통 결합의 결점
  - 첫째, 변수 값의 초기화 시기를 각 모듈이 미리 알고 있어야 함
  - 둘째, 전역 변수는 모듈 사이에 정보 교환에 사용




```
char character;      /* global variable */

Read_and_Print()
{
        .....
        Read_char();
        Print_char();
        .....
}
Read_char()
{
        .....
        character := getchar();
        .....
}
Print_char()
{
        .....
        putchar(character);
        .....
}
```

내용 결합도(Content Coupling)

- 한 모듈이 다른 모듈 내부에 있는 데이터나 제어 신호를 사용하는 경우 또는 다른 모듈의 중간에 뛰어 들어가는 (Branching) 경우
- 모듈들이 서로 상대방의 내부 정보를 직접 이용한다는 것은 모듈화가 잘 되어있지 않다는 것으로 소프트웨어의 설계에서 극소수 상황을 제외하고는 이런 연결은 피하여야 함
- 현재 우리가 쓰고 있는 고급 프로그래밍 언어(High Level Programming Language)는 이런 결합을 모듈들의 관계에서 지원하지 않음

  예) 한 모듈에서 GOTO 문으로 다른 모듈 내부에 들어 갈 수 없고, 타 모듈 내부에서 정의된 변수도 이용할 수 없음

  - FORTRAN 언어 : ENTRY 문을 이용하여 다른 모듈의 내부에 직접 접근할 수 있으며, 프로그램 성능을 고려하여 어셈블리 언어를 이용하여 내부 결합도를 가진 모듈을 설계할 수 있음

12.4.4 이해도(Understandability)

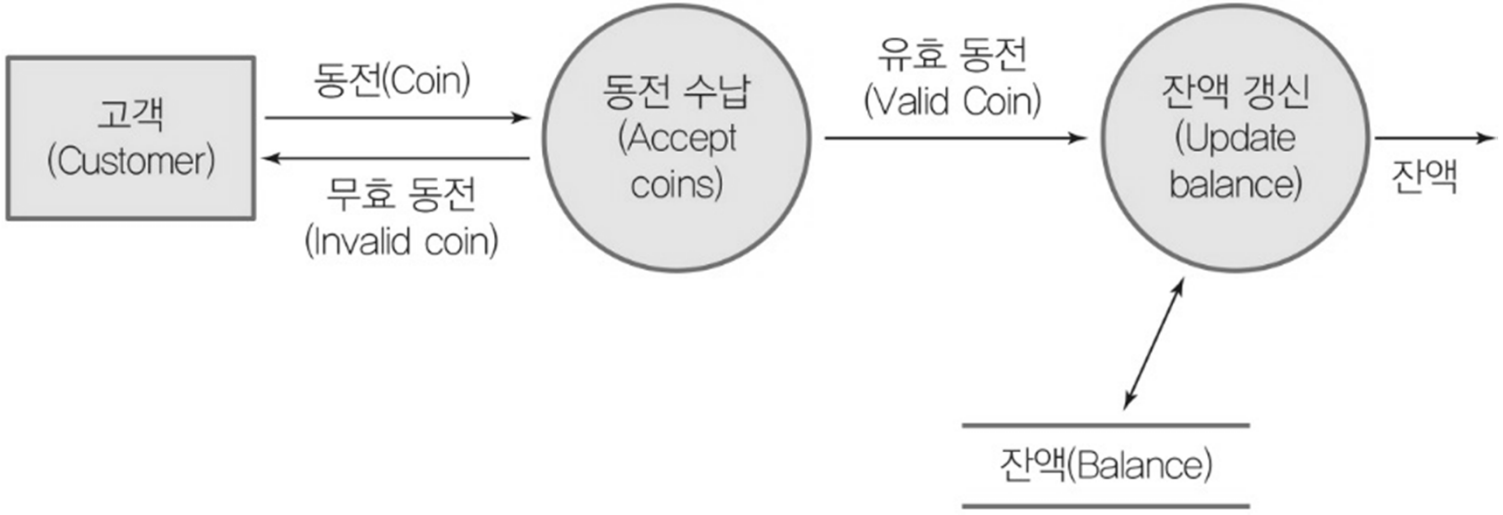
- 프로그램 요소의 동작을 이해하지 않고는 프로그램을 만들기도 어려우며 유지보수 단계에서 수정이 어려움
- 프로그램 및 문서의 **이해도**(Understandability)는 다른 프로그램 요소나 정보를 참조하지 않고 이해할 수 있는 용이성
- 시스템 응집도가 높을수록 프로그램 요소들을 쉽게 이해할 수 있으며 고쳐나가기 쉬움

이해도와 관계 있는 요소들

- 구조적으로 소프트웨어를 설계해야 하며, 시스템은 서브시스템으로 구성되고 각 서브시스템은 독립성이 높아야 함
- 서브시스템을 구성하는 모듈 간(결합도)의 독립성을 높여야 하며, 모듈들은 서로 낮은 결합도를 갖고, 정보 은닉이 잘되어져 서로 참조 정보 최소화
- 프로그램 이해의 전제는 프로그램을 읽는 것이기 때문에 프로그램을 읽기 쉽게 작성하여야 하고, 프로그램 구조, 모듈 및 변수 이름, 모듈의 복잡성 등은 프로그램을 쉽게 읽는 정도(Readability)에 많은 영향

이해도와 관계 있는 요소들 (계속)

- 설계 및 코드 문서는 정확하고 완전해야 함.
  - 부정확한 문서는 프로그램 이해에 도움을 주지 못할 뿐만 아니라 더욱 어렵게 함
  - 문서가 없거나 불완전하면 코드에 의해서만 프로그램을 이해해야 하므로 프로그램 이해가 어려움



12.4.5 적응도(Adaptability)

- 새로운 환경에 적절히 대응할 수 있도록 소프트웨어를 변경시킬 수 있는 용이성
- 소프트웨어의 경우 운용 환경의 변화는 피할 수 없는 것
- 적응도가 높은 시스템을 만들기 위해 각 구성요소 간 결합도가 낮아야 함
- 문서들은 이해하기 쉬워야 하고 프로그램과 일치하도록 관리
- 환경 독립성을 높여 수정되어야 할 부분을 지역화하여 적은 부분을 수정하여 쉽게 새로운 환경에 적응할 수 있도록 해야 함
- 환경과 연관된 부분을 지역화하여 이해도를 높이고 이식성(Portability)도 높여야 함

12장 요약

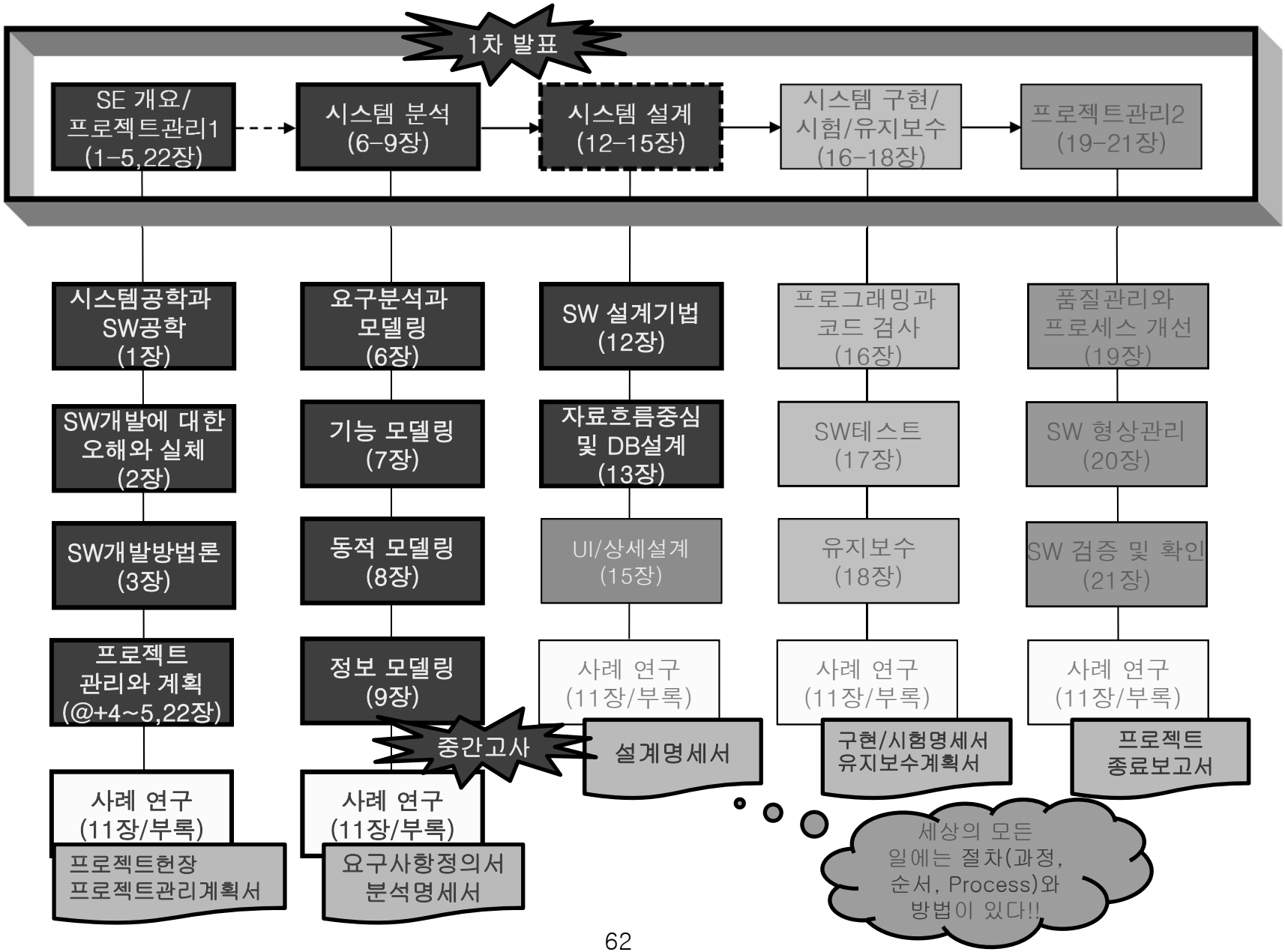
- 소프트웨어 설계는 소프트웨어 개발의 핵심
- 사용자 요구사항 분석은 안정감 있는 시스템을 설계하기 위한 준비 과정
- 요구사항에 대하여 보다 잘 이해할수록 이상적인 설계 가능
- 모듈들은 서로 독립적이어야 하고, 각 구성요소는 내부의 응집력이 높아야 함
- 모듈들 사이의 연결을 나타내는 결합도 최소화
- 주요 설계 활동
  - 기술적 측면 : 데이터 설계, 구조 설계, 절차 설계, 인터페이스 설계
  - 관리적 측면 : 기본 설계, 상세 설계
- 소프트웨어 품질
  - 설계에서 시작
  - 설계는 소프트웨어 품질을 향상시킬 수 있는 중요한 과정
  - 요구사항을 소프트웨어 시스템으로 변형시키는 개발의 첫단계

계획~분석 단계 종료 및 설계 단계 진행 준비 확인

- 계획~분석 단계 종료 확인 : 산출물 및 인수인계
  - 1차 발표 결과 반영 + 신논리 모델(추가 요구사항 반영) + 요구사항 추적성 매트릭스 작성 후 요구사항 분석 완료
  - 이전 단계(계획~분석) 산출물 수정 보완(피드백)
    - 프로젝트 변경사항(인원, 일정, 과업범위 등)을 반영하여 계획 수정
  - 팀장 변경(4차) 및 인수인계

- 설계 단계 진행 준비 확인: 업무 표준화/팀별 할당/PM지정
  ① 팀별로 업무분류표 작성
  ② 반별 통합회의 실시(10/11주차 실시)
  ③ 업무분류표 통합 및 표준화(단일 시스템, 표준 업무분류표 작성)
  ④ 표준 업무분류표를 기준으로 팀별 설계 범위 할당(1/N, N=팀수)
  ⑤ 프로젝트 관리 방법 조정(팀 단위 ⇒ 전체/분반통합)
  ⑥ 전체 프로젝트 통합 관리자 선정(분반장에서 1명, 팀장은 팀별 프로젝트 관리자 및 프로젝트 파트 리더 역할 수행)

강의 로드맵(12주차)


1차 발표
SE 개요/
프로젝트관리1
(1-5,22장)
시스템 분석
(6-9장)
시스템 설계
(12-15장)
시스템 구현/
시험/유지보수
(16-18장)
프로젝트관리2
(19-21장)
시스템공학과
SW공학
(1장)
SW개발에 대한
오해와 실체
(2장)
SW개발방법론
(3장)
프로젝트
관리와 계획
(@+4~5,22장)
사례 연구
(11장/부록)
프로젝트헌장
프로젝트관리계획서
요구분석과
모델링
(6장)
기능 모델링
(7장)
동적 모델링
(8장)
정보 모델링
(9장)
사례 연구
(11장/부록)
요구사항정의서
분석명세서
SW 설계기법
(12장)
자료흐름중심
및 DB설계
(13장)
UI/상세설계
(15장)
사례 연구
(11장/부록)
중간고사
설계명세서
프로그래밍과
코드 검사
(16장)
SW테스트
(17장)
유지보수
(18장)
사례 연구
(11장/부록)
구현/시험명세서
유지보수계획서
품질관리와
프로세스 개선
(19장)
SW 형상관리
(20장)
SW 검증 및 확인
(21장)
사례 연구
(11장/부록)
프로젝트
종료보고서
세상의 모든
일에는 절차(과정,
순서, Process)와
방법이 있다!!

강의 계획 피드백(12주차)

3. 강의계획				
주차	강의주제	강의내용	과제	평가
1주차	SW공학 개요(1)	과목 소개, 시스템공학과 소프트웨어공학	#1 소프트웨어 개발 관련(5)	
2주차	SW공학 개요(2)	SW 개발 실체 및 개발방법론	팀 편성 및 프로젝트 과제 선정	
3주차	프로젝트 관리 이해(1)	프로젝트 관리(1)- 프로세스와 프로젝트 계획	#2 프로젝트 계획서(5)	
4주차	프로젝트 관리 이해(2)	프로젝트 관리(2)- 원가, 일정, 비용 관리		
5주차	요구사항 분석(1)	요구사항 추출 및 모델링 방법	#3 요구사항 정의서(5)	
6주차	요구사항 분석(2)	구조적 분석기법(1) 및 기능모델링		
7주차	요구사항 분석(3)	구조적 분석기법(2) 및 동적/정보 모델링	#4 분석 명세서(5)	
8주차	설계(1)	설계 개요 및 주요 설계기법		
9주차	중간고사 및 중간발표	중간고사/중간발표 평가(계획, 분석)	#5 중간발표(계획-분석,10)	중간고사(10)
10주차	설계(2)	구조적 설계기법(1)-자료흐름중심 설계		
11주차	설계(3)	구조적 설계기법(2)- DB/사용자 인터페이스/상세 설계	#6 설계 명세서(5)	
12주차	구현 및 시험	프로그래밍 및 코드검사, 시험 절차/기법, 구현/시험 계획	#7 구현/시험 명세서(5)	
13주차	인도(전환) 및 유지보수/ 프로세스 개선	인도(전환)/유지보수 및 SW 프로세스 개선, 인도(전환)/유지보수 계획	#8 인도/유지보수 계획서(5)	
14주차	프로젝트 관리 이해(3)	품질/형상관리, 검증 및 확인, 종료 보고, 품질/형상관리계획 및 종료보고	#9 프로젝트 종료 보고서(5)	
15주차	기말고사 및 최종발표	기말고사/최종발표 평가(설계/구현/시험/유지보수/종료)	#10 최종발표(설계-종료,10)	기말고사(10)/팀 및동료평가(10)

다음 주(13주차) 강의계획

- 교재 12, 13장 연습문제 풀어보기
- 요구사항 분석 마무리 및 시스템 구조 설계 준비
- 계획~분석 최종 산출물 베이스라인 **설정(승인)**
- 설계단계 진행 : 시스템 구조 설계(기본/개략 설계) 진행
- 교재 15, 16장 읽어오기

☞ 다음 주(13주차)는 **교재 15~16장 강의 예정**