7강. 소프트웨어 설계

컴퓨터과학과 김희천 교수

목차

Chapter. 1

설계 개요

1. 소프트웨어 설계

설 계

- 물건을 만들기 위한 계획 또는 제작을 위해 물건을 의미 있게 표현하는 것
- 계획이나 표현은 요구사항을 만족하고 품질이 좋아야 함

✚ 소프트웨어 설계

– 외부에서 관찰 가능한 행위 명세(요구사항)에 구현을 위한 방법을 명시하는 것

2. 설계 프로세스(1/2)

✚ **설계 작업의 입력**

- 요구사항과 분석 모델, 환경적 제약사항

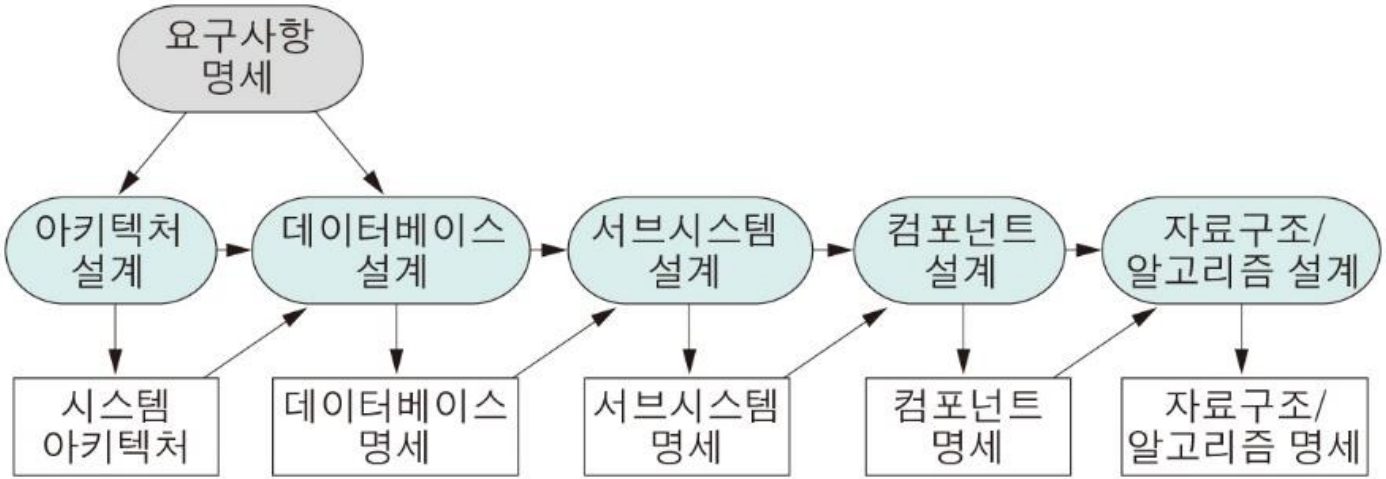
✚ **설계 작업의 결과물**

- 구성 요소들의 관계를 보여주는 소프트웨어 아키텍처
- 요소들의 인터페이스와 알고리즘
- 데이터에 관한 정의

✖ **설계 프로세스는 반복적 프로세스로**
초기 버전을 만들고 지속적으로 수정하면서
점점 정형화를 가하고 세세하게 기술함

- 결과는 추상화 수준이 다른 여러 모델로 나타남

2. 설계 프로세스(2/2)


요구사항
명세
아키텍처
설계
데이터베이스
설계
서브시스템
설계
컴포넌트
설계
자료구조/
알고리즘 설계
시스템
아키텍처
데이터베이스
명세
서브시스템
명세
컴포넌트
명세
자료구조/
알고리즘 명세

3. 설계 원리- 문제의 분할(1/3)

분할 후 정복

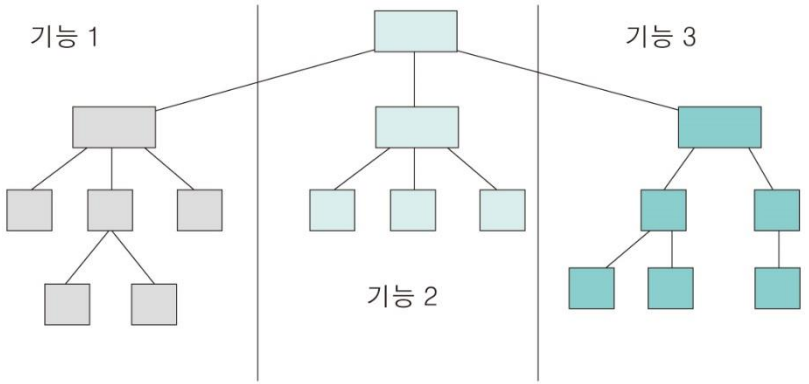
문제를 작은 것들로 나누고 각각을 독립적으로 정복하는 것

✚ 수평 분할

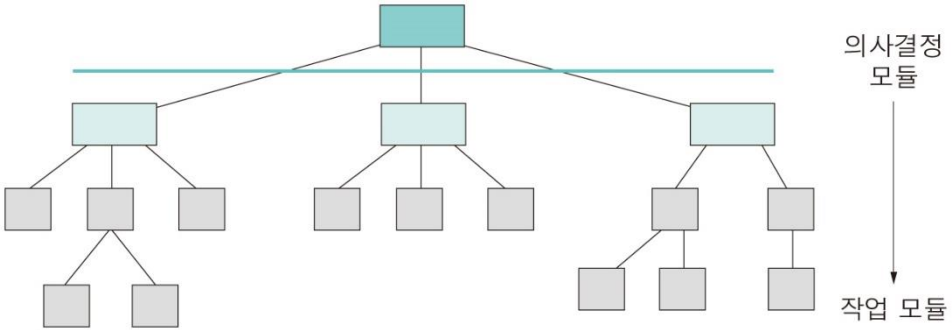
- 기능들을 분리된 가지에 할당
- 단순한 형태는 입력 작업, 데이터 변환, 출력 작업을 담당하도록 분리
- 테스트, 유지보수, 확장이 용이하고 부작용의 파급효과가 줄어듦
- 많은 데이터가 모듈 인터페이스를 통해 이동되어 전체 제어가 복잡해 짐

3. 설계 원리- 문제의 분할(2/3)

✚ 수평 분할



✚ 수직 분할



3. 설계 원리- 문제의 분할(3/3)

수직 분할

- 제어 모듈과 작업 모듈을 위와 아래로 분산함
- 상위 수준의 모듈들은 제어 기능을,
  하위의 모듈들은 실제 작업(입출력과 데이터 변형 작업)을 수행
- 작업 모듈이 변경되기 쉬우며, 제어 모듈이 변경되면 부작용이 큼

4. 설계 원리- 추상화(1/2)

추상화

- 내부의 상세한 내용을 생략하고
  외부 행위만을 기술하는 것

✚ 추상화의 적용

- 시스템 설계에서 구성 모듈들을 추상화하여 표현함
- 추상적으로 표현된 다른 모듈이 존재하므로 상세 설계 단계에서 하나의 모듈에 집중할 수 있음
- 추상화된 시스템 모델을 통해 시스템의 분석이 가능함

4. 설계 원리- 추상화(2/2)

기능 추상화

- 수행하는 기능으로 모듈을 명세하는 것
- 전체를 작은 기능들로 분해하는 것(기능 중심의 분할 방법)

데이터 추상화

- 데이터와 데이터 조작에 필요한 오퍼레이션을 함께 묶는 것
- 객체의 상세한 구현 내용이 감추어지며 데이터 조작에 공개된 오퍼레이션만을 사용해야 함

5. 설계 원리- 하향식과 상향식 설계

✚ 하향식 설계

- 계층 구조상에서 시스템의 주요 컴포넌트들을 찾고 그것을 낮은 수준의 컴포넌트들로 분해하는 것
- 단계적 정제라 하며 메인 모듈의 설계에서 시작하여 단계적으로 구체화시키는 것

✚ 상향식 설계

- 가장 기본적인 컴포넌트를 먼저 설계하고 이것을 사용하는 상위 수준의 컴포넌트를 설계함

✕ 고려 사항

- 시스템 명세가 명확한 경우나 모든 것을 새로 개발하는 작업에는 하향식이 적합
- 기존 컴포넌트들을 조합하여 시스템을 개발하는 경우에는 상향식이 적합

Chapter. 2

아키텍처 설계

1. 아키텍처 설계와 중요성

아키텍처 설계

- 시스템의 주요 기능을 제공하는 구성 요소들을 식별하고 이들 간의 관계를 정하는 것(초기 설계)
- 시스템의 품질 특성과 제약사항을 반영해야 함

✚ 아키텍처의 중요성

- ✕ 개발이 진행된 후에는 시스템 구조를 수정하기 어려움
- ✕ 요구 명세 활동 또는 설계 초기에 이루어지며 개발 과정에 큰 영향을 줌
- ✕ 주요 구성 요소들을 개별적으로 명세하기 위한 출발점
- ✕ 프로젝트 참여자들 사이의 중요한 의사소통 수단

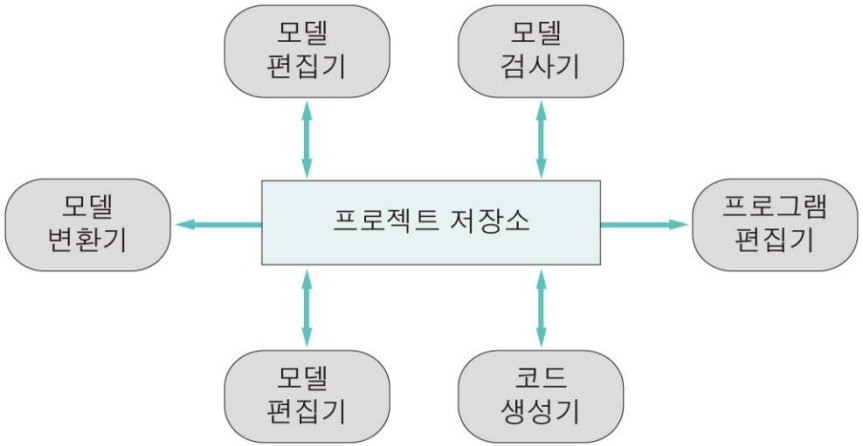
2. 아키텍처 스타일

- 유사한 애플리케이션들에서 사용되는 공통적인 아키텍처 패턴
- 미리 만들어진 시스템 설계 모델로 아키텍처 설계의 초안이 됨
- 아키텍처 스타일에 따라 적합한 응용 분야가 있으며 장단점이 알려져 있음

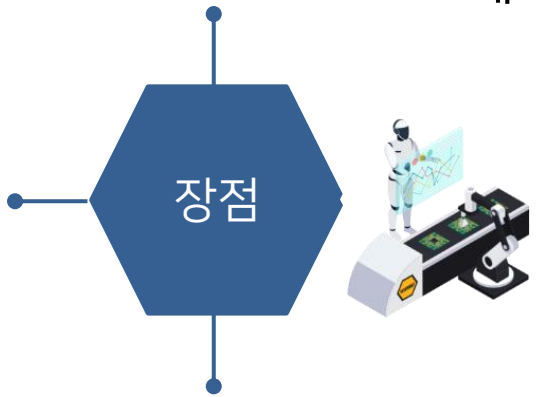
3. 데이터 중심 아키텍처(1/2)

- **저장소 모델**이라 하며 서브시스템 간
  대규모 데이터를 공유하는 소프트웨어 시스템에 사용됨

- 공유된 데이터베이스에 기반한 아키텍처로 저장소를 통해 상호 작용함
  - 서브시스템들은 중앙에서 통합 관리되는 데이터베이스를 통해 상호작용 함



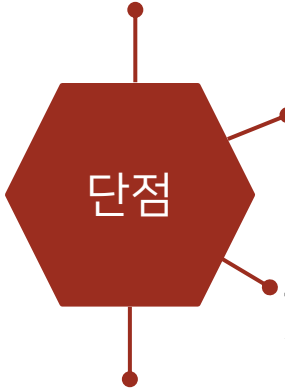
3. 데이터 중심 아키텍처(2/2)



데이터 관리가 효율적

데이터를 생성하는 서브시스템과 데이터를 사용하는 서브시스템이 서로 독립적임

공유 데이터 모델을 사용하는 경우 시스템의 확장이 용이함



서브시스템은 공통의 데이터 모델을 사용해야 함

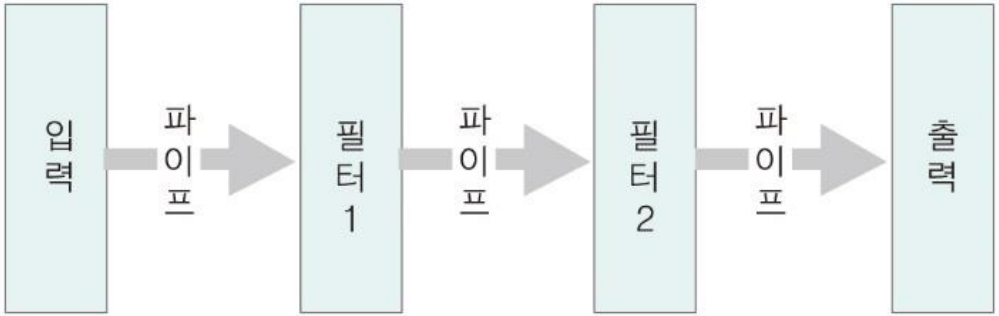
중앙 저장소의 성능이 저하되면 병목 현상이 발생함

모든 서브시스템에 동일한 정책이 적용됨

데이터 모델을 변경하면 큰 비용이 발생함

4. 데이터 흐름 아키텍처

+ **파이프 필터 구조**라고 하며 데이터 요소들에 대한 개별 변환 작업들을 연결하여 시스템을 구성
  - ×서브시스템은 필터에 해당하며 입력 데이터를 받아 처리하고 다른 곳으로 출력함
+ **사용자 간섭 없이 데이터 스트림에 일련의 변환 작업을 적용하는 시스템에 적합함**
  - ×구성 요소들 간의 복잡한 상호 작용이 요구되는 경우에는 적합하지 않음



5. 클라이언트-서버 아키텍처(1/2)

✚ 특정 서비스를 제공하는 서버의 집합과
서비스를 이용하는 클라이언트들로 시스템을 구성

– 분산 시스템의 아키텍처로 서버와 클라이언트는 네트워크로 연결됨

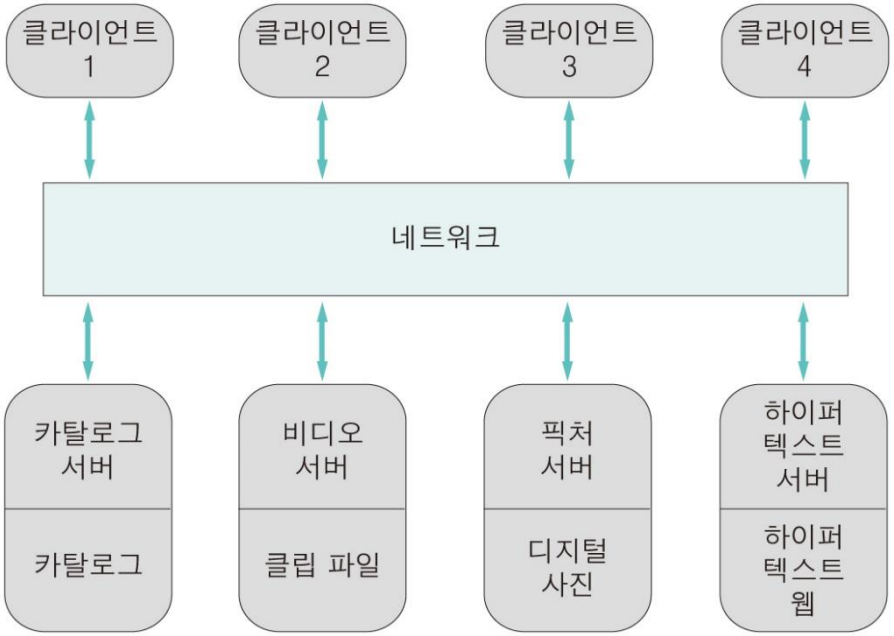
✕장점

- 데이터와 데이터 처리가 분산됨
- 분산된 프로세서들을 효율적으로 사용할 수 있음
- 새로운 서버를 추가하기 쉬움

✕단점

- 공유 데이터 모델이 없으므로 데이터의 교환이 비효율적
- 모든 서버가 각각 데이터 관리 책임을 가짐

5. 클라이언트-서버 아키텍처(2/2)



6. 계층형 아키텍처

- 추상 기계 모델이라 하며 시스템을 계층별로 구성
- 하위 계층이 제공하는 서비스를 상위 계층이 이용
  - 가장 외부(상위) 계층은 사용자 인터페이스 기능을 제공하고
    가장 내부(하위) 계층은 하드웨어와의 상호 작용을 제공

✕특징

- 각 계층은 특정 서비스를 제공하며
  서브시스템들의 점증적 개발이 가능
- 특정 계층의 인터페이스가 변경되면
  인접 계층만 영향을 줌
- 실제로는 시스템을 계층 구조로 만들기 어려움

7. MVC 아키텍처

- 시스템의 기능을 도메인 지식을 관리하는 모델(M), 그것을 사용자에게 보여주는 뷰(V), 사용자와의 상호작용을 제어하는 컨트롤러(C)로 분리
  - 컨트롤러는 사용자에게 입력을 얻어 모델에 알리고, 모델은 데이터를 관리하며 변경이 일어날 때 뷰에게 통지하고, 뷰는 모델로 부터 데이터를 가져와 디스플레이 함
  - 모델은 데이터를 관리하며, 뷰의 변경이 모델에 영향을 주지 않음

- 고려 사항
  - 저장소 모델의 특수한 형태
  - 하나의 모델에 대해 여러 뷰를 제공해야 하는 대화형 시스템 또는 웹기반 시스템에 적합

8. 3계층 아키텍처

✚ 클라이언트-서버 모델의 개선된 형태

✚ 사용자 인터페이스, 애플리케이션 로직, 저장소의 3계층으로 분리하여 시스템을 구성

- 사용자 인터페이스 계층은 사용자와의 상호작용을 담당
- 애플리케이션 로직 계층은 애플리케이션이 필요로 하는 기능을 담당
- 저장소 계층은 데이터 관리를 담당

✖ 고려 사항

- 인터페이스 계층과 애플리케이션 로직 계층을 분리함으로써 같은 애플리케이션에 대해 다른 사용자 인터페이스의 개발이 가능함
- 애플리케이션 로직이 자주 바뀌는 경우 대응이 용이함

9. 아키텍처와 비기능적 속성

- 소프트웨어 아키텍처는 비기능적 요구사항에 큰 영향을 줌
- 비기능적 속성을 고려하여 설계해야 함
  - 성능: 서브시스템들 간의 통신을 최소화
  - 보안: 계층형 아키텍처를 사용하고 보안이 중요한 시스템을 내부 계층에 위치시킴
  - 안전성: 안전성이 요구되는 요소들을 하나의 서브시스템에 집중시킴
  - 가용성: 동일 기능의 컴포넌트를 중복시킴
  - 유지보수성: 자료의 공유를 피하고 모듈화시켜야 함
- 모순되는 점이 존재하므로 타협이 필요함
  - 성능을 위해서는 큰 규모의 컴포넌트가 좋으나 유지보수성이 떨어짐
  - 중복된 데이터를 유지하는 것은 가용성을 좋으나 보안 유지가 힘듦

Chapter. 3

구조적 설계

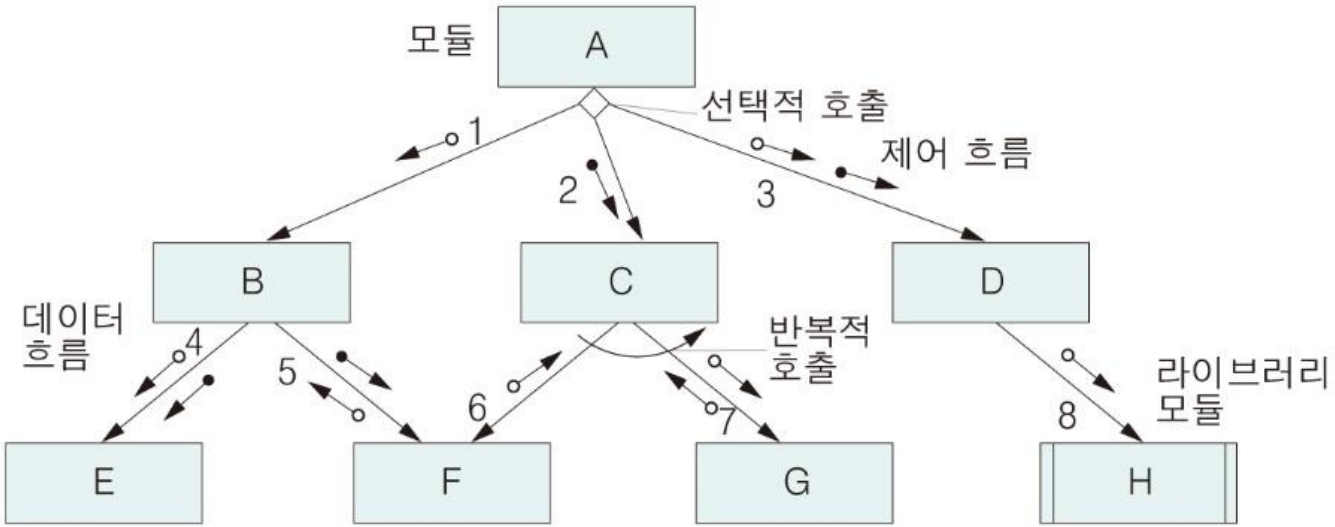
1. 구조적 설계

- 구조적 분석(SA)의 결과물을 이용해 아키텍처를 설계하고 모듈을 개발하는 방법
  - 아키텍처 표현을 위해 구조도를 사용함
  - 구조적 설계에서 아키텍처 설계는 데이터 흐름도(DFD)에 표현된 정보의 흐름을 이용하여 구조도(SC)로 변환하는 작업
  - 구조도는 모듈들의 계층 구조와 제어 구조를 보여줌
- 결과물은 구조도, 모듈 명세서, 자료 사전

2. 구조도(1/2)

- 모듈들의 계층 구조, 모듈의 매개 변수, 모듈들 간의 상호 연결 관계를 보여줌
  - 모듈들은 블랙박스로 표현되며 계층적으로 배열됨
  - 상위 모듈이 하위 모듈을 호출함(위에서 아래로, 좌에서 우로 해석됨)
- 구조도의 구성 요소
  - 모듈, 라이브러리 모듈
  - 데이터 흐름, 제어 흐름
  - 호출, 선택적 호출, 반복적 호출

2. 구조도(2/2)



3. 데이터 흐름도를 구조도로 변환하는 방법

① 데이터 흐름의 유형을 선정(변환 흐름과 트랙잭션 흐름)

② 흐름 경계선을 파악

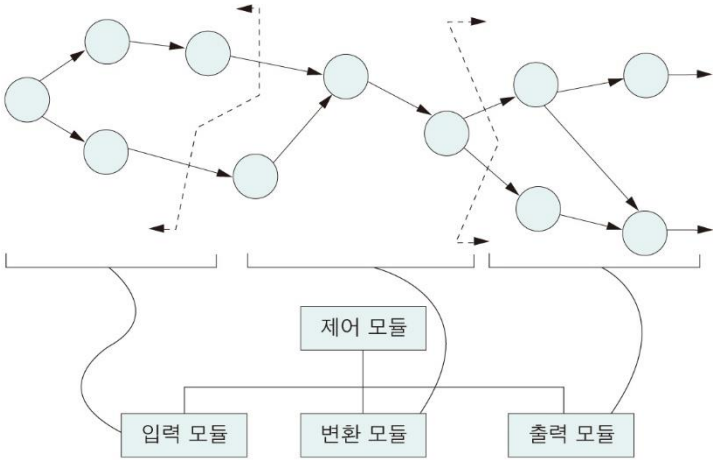
③ 데이터 흐름도를 구조도로 변환

④ 제어 계층을 정의

⑤ 산출된 구조도를 검토하고 다듬음

4. 변환 분석에 의한 구조적 설계

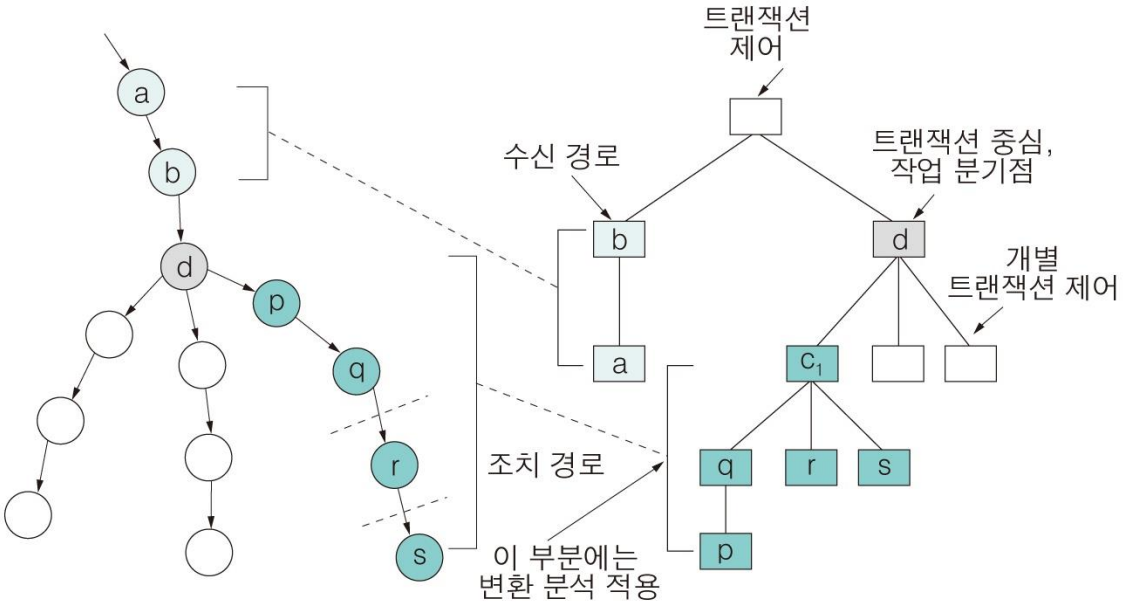
- 데이터 흐름의 유형이 변환 흐름인 경우 사용
- 변환 흐름은 데이터를 입력 받고, 변환/가공하고, 결과를 출력하는 유형
- 변환 분석
  - 데이터 흐름도에서 입력 흐름과 출력 흐름의 경계를 표시
  - 구조도에 입력, 변환, 출력 모듈을 제어하는 모듈을 추가로 만듦



5. 트랜잭션 분석에 의한 구조적 설계(1/2)

- 데이터 흐름의 유형이 트랜잭션 흐름인 경우 사용
- 트랜잭션 흐름은 한 프로세스에서 입력을 여러 경로의 데이터 흐름으로 유출하는 유형
  - 트랜잭션 중심 모듈은 트랜잭션의 유형을 결정하고 적당한 모듈을 호출하는 모듈
- 트랙잭션 분석
  - 트랜잭션의 소스를 식별하고 트랜잭션 중심 모듈을 찾음
  - 전체를 제어하는 모듈을 추가함
  - 데이터 수신 경로와 트랜잭션 중심에 해당하는 모듈을 하위에 위치 시킴
  - 처리 부분에 있는 프로세스를 모듈로 변환함

5. 트랜잭션 분석에 의한 구조적 설계(2/2)



Chapter. 4

상세 설계와 모듈화

1. 상세 설계

✚ 개별 모듈에서 사용되는 자료 구조와 알고리즘을 자세히 설계

– 상세 설계는 기능적 요구사항에 초점을 맞춤

✕ 아키텍처 설계와의 비교

- 아키텍처 설계는 비기능적 요구사항에 초점을 두고 아키텍처 스타일, 컴포넌트 재사용, 설계 원리, 설계 제약사항을 고려함

2. 모듈화

✚ 시스템을 여러 분리된 컴포넌트들로 나누어 구성하는 것
- 시스템을 구현할 때, 각 컴포넌트별로 분리하여 구현할 수 있음
- 컴포넌트의 변경으로 인한 파급효과가 최소화됨

✚ 모듈화된 시스템은 잘 정의되고 관리 가능한 단위들로 구성되며, 각 단위들은 잘 정의된 인터페이스를 통해 소통함

✕ 장점
- 구성 요소들이 기능적으로 독립적이어서 이해하기 쉬움
- 구성 요소들을 독립된 단위로 문서화할 수 있음
- 작고 간단한 문제에 집중할 수 있음
- 테스트와 디버깅, 유지보수 작업이 쉬움
- 재사용하기 용이함

3. 모듈의 독립성

모듈의 독립성으로 시스템 설계 결과를 평가할 수 있음

- 기능이 분리될 수 있고, 인터페이스 관계가 단순해 지며, 다른 모듈과의 상호 작용이 최소화됨
- 따라서 개발 작업과 테스트 작업이 쉬워지고 수정할 때 파급 효과가 작아짐

모듈 독립성의 평가

- 하나의 모듈은 높은 응집력을 가져야 함
- 다른 모듈과의 결합도가 느슨해야 함

4. 결합도

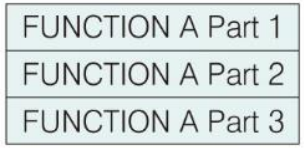
- **두 모듈 사이의 상호 의존성 정도를 의미하며 두 모듈을 연결하는 인터페이스의 복잡도에 좌우됨**
  - 데이터의 수와 유형, 공유 데이터의 사용 여부, 제어 흐름의 전달 여부 등을 판단
- **결합도의 수준**
  - 데이터 결합: 기본 유형의 데이터 (가장 느슨함)
  - 스탬프 결합: 구조체와 같은 복합 자료
  - 제어 결합: 제어 플래그
  - 공통 결합: 전역 변수를 공유
  - 내용 결합: 코드의 공유

5. 응집력

하나의 모듈이 가지는 기능적 집중성에 관한 척도

- 모듈을 구성하는 요소들이 기능적으로 얼마나 관련성이 있는가

응집력의 수준



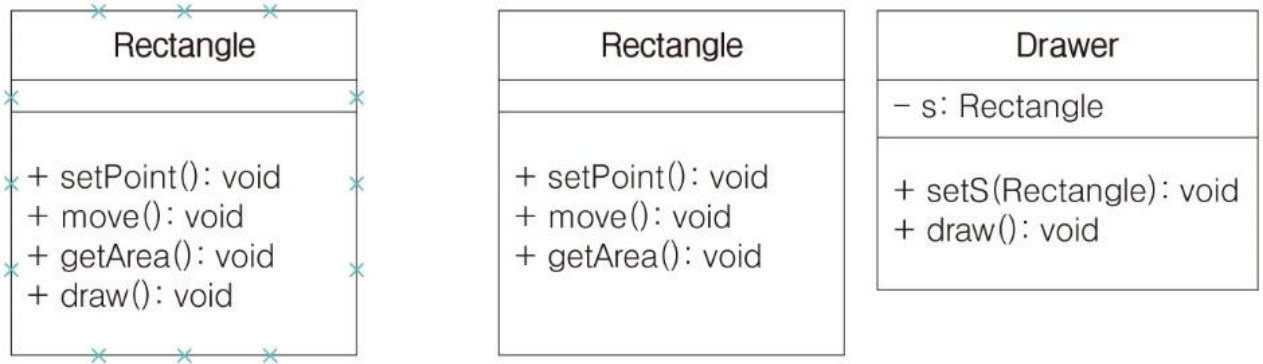
- 기능적 응집력: 요소들이 단일 작업을 수행하기 위해 협력(가장 강함)
- 순차적 응집력: 한 요소가 출력한 것이 다른 요소의 입력이 됨
- 통신 응집력: 수행하는 기능들이 같은 자료 구조를 다룸
- 절차적 응집력: 특정 순서를 따르는 기능들
- 시간적 응집력: 같은 시간 주기에 실행되는 기능들
- 논리적 응집력: 같은 범주의 기능들(모든 출력을 수행하는 모듈)
- 우연적 응집력: 연관성이 없는 기능들

Chapter. 5

객체지향 설계 원칙 - SOLID

1. 단일 책임 원칙 - SRP

- 모든 클래스는 하나의 책임만 가져야 한다.
  - 유일한 이유로만 변경될 수 있다.

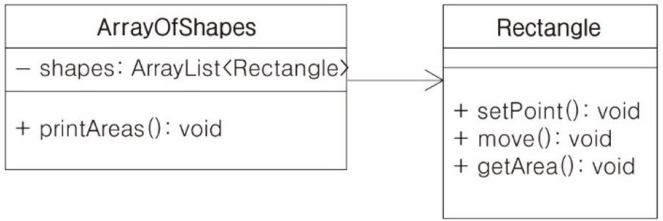


(a) 적용 전 (b) 적용 후

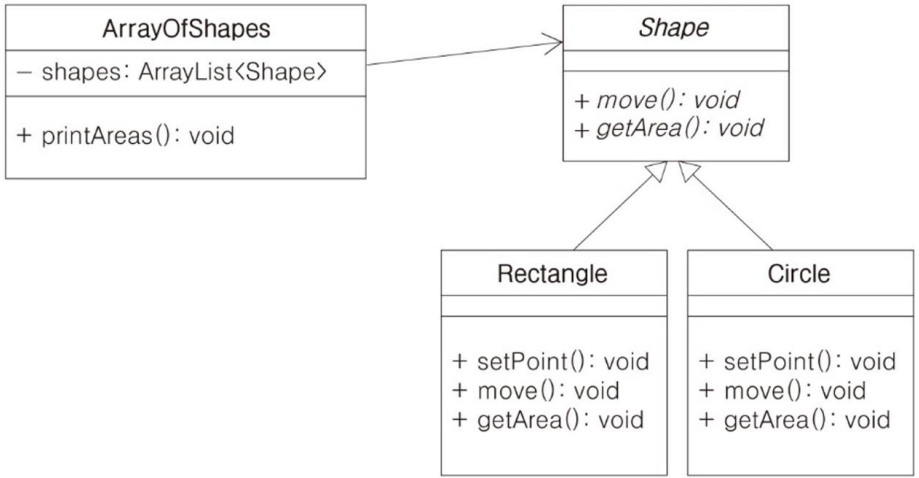
2. 개방 폐쇄의 원칙 - OCP

✚ 클래스가 확장에는 열려 있고, 수정에는 닫혀 있어야 한다.

✕새로운 기능을 추가하면서도 기존 코드를 수정하지 않아도 된다.



(a) 적용 전



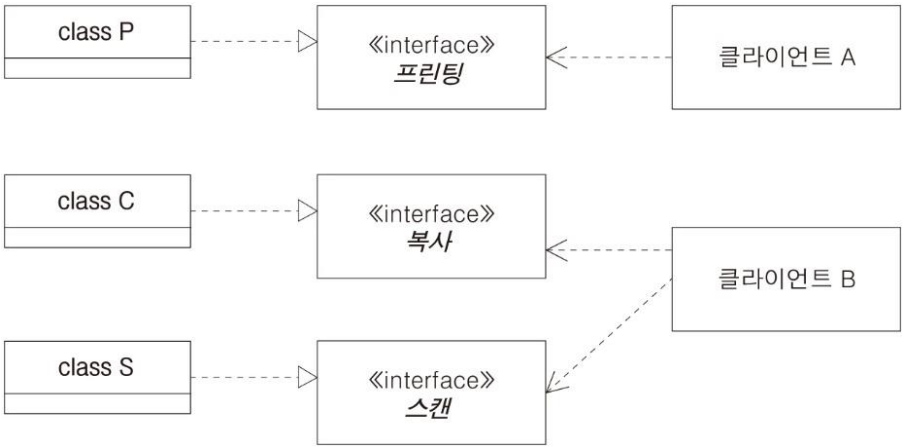
(b) 적용 후

3. 리스코프 교체의 원칙 - LSP

- **프로그램에 존재하는 부모 유형(T)의 객체를 자식 유형(S)의 객체로 교체할 수 있어야 함**
  - T와 S 간 상속 관계가 적절해야 한다는 의미
  - LSP가 만족되면 새로운 자식 클래스를 추가할 때 기존 코드의 수정이 필요 없음
  - 자식 클래스에서 부모 클래스의 메소드를 재정의하지 않으면 LSP의 문제가 없음
  - 자식 클래스에서 부모 클래스의 메소드를 재정의하는 경우는?

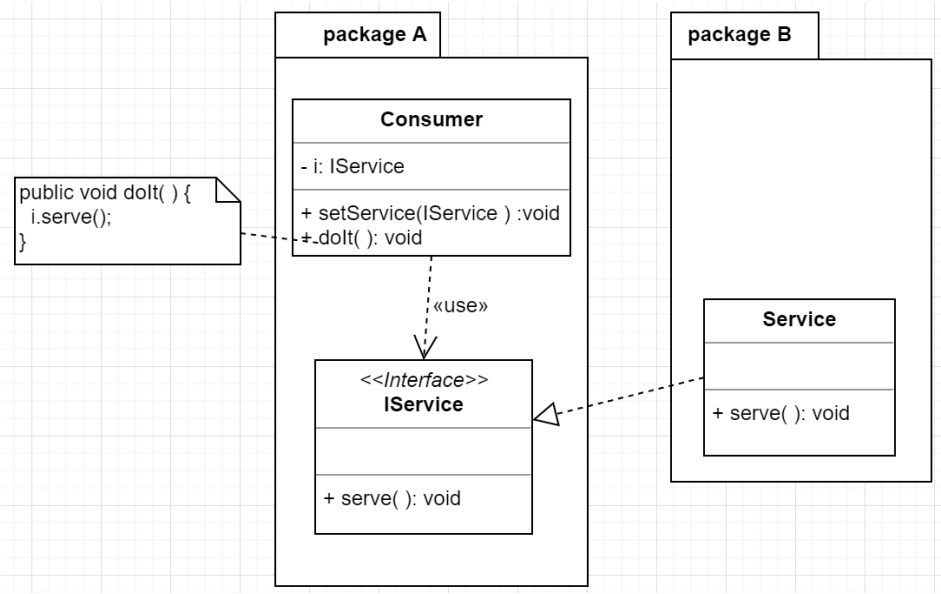
4. 인터페이스 분리의 원칙 - ISP

- 하나의 큰 범용 인터페이스가 아닌
  작고 특화된 여러 인터페이스로 나누어 설계할 것
  - ISP를 만족하지 못한다면, 사용하지 않는 인터페이스가 바뀔 때도 클라이언트에게 소프트웨어를 재배포 해야 함
  - 복합기 인터페이스 대신에 프린팅, 복사, 스캔 인터페이스로 나눌 것



5. 의존 관계 역전의 원칙 - DIP

- 구체적 저수준 사물(구체 클래스)이 아닌 추상적 고수준 개념(인터페이스)에 의존해야 함
  - 실제 작업을 수행하는 저수준 구체 클래스는 변경되기 쉬움



다음강의

8강. 소프트웨어 유지보수