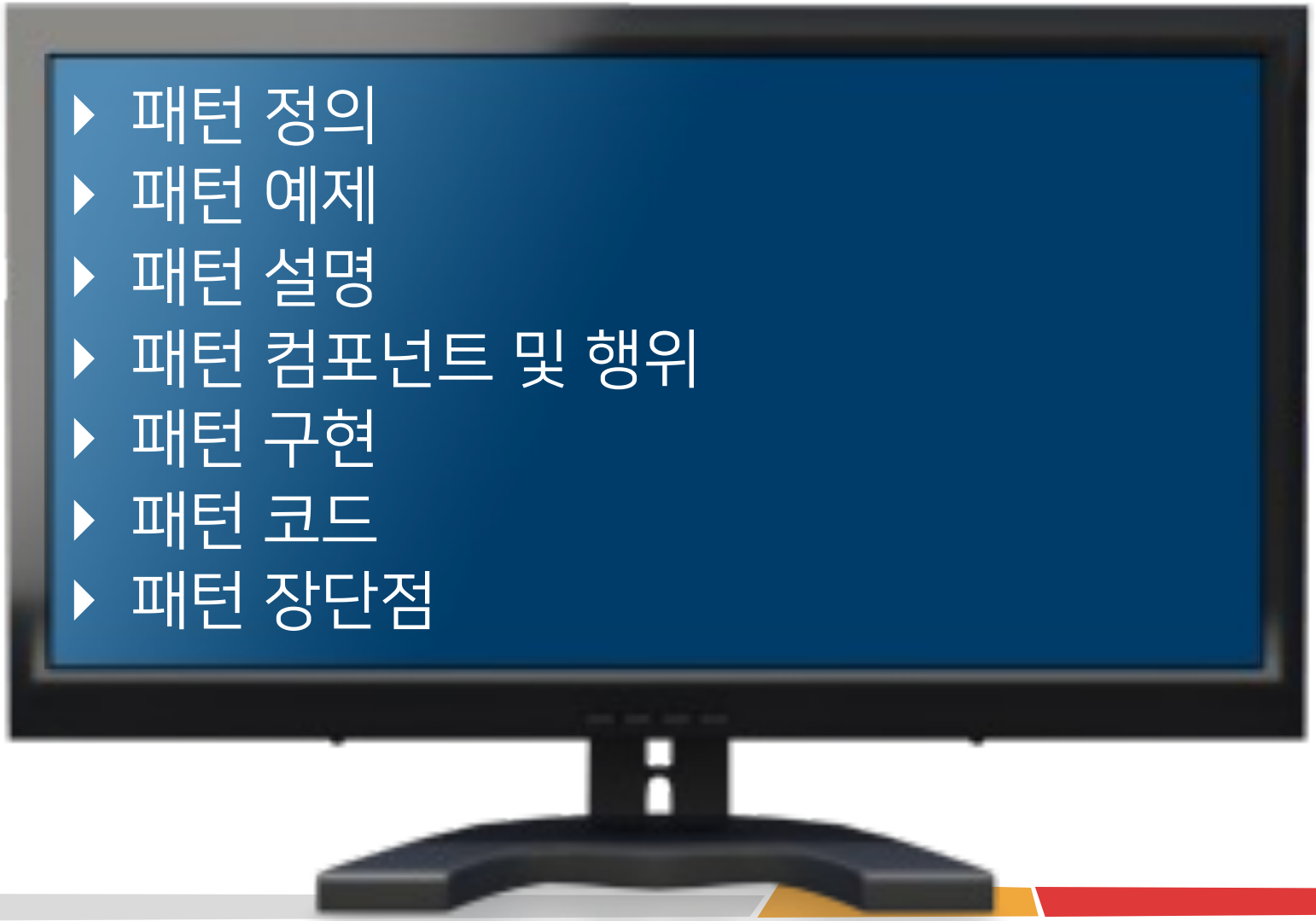
Spring 2023

# 소프트웨어 아키텍처 패턴: Pipe-and-Filter

Seonah Lee

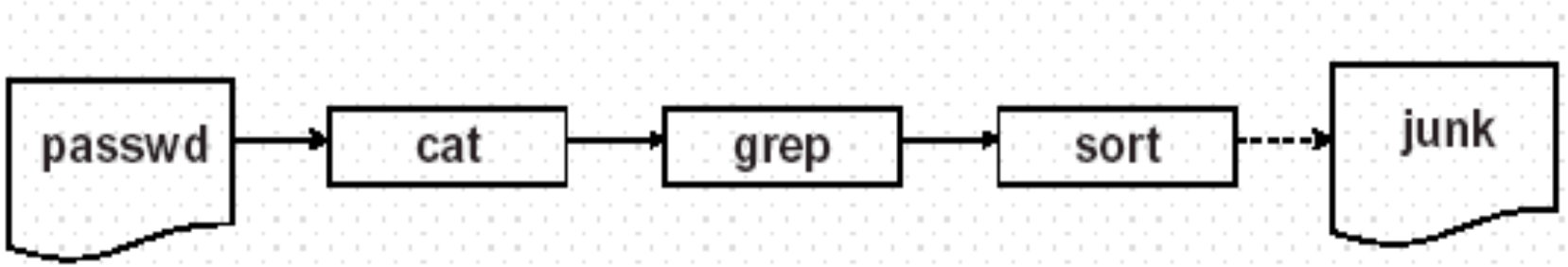
Gyeongsang National University

# Pipe-and-Filter 패턴



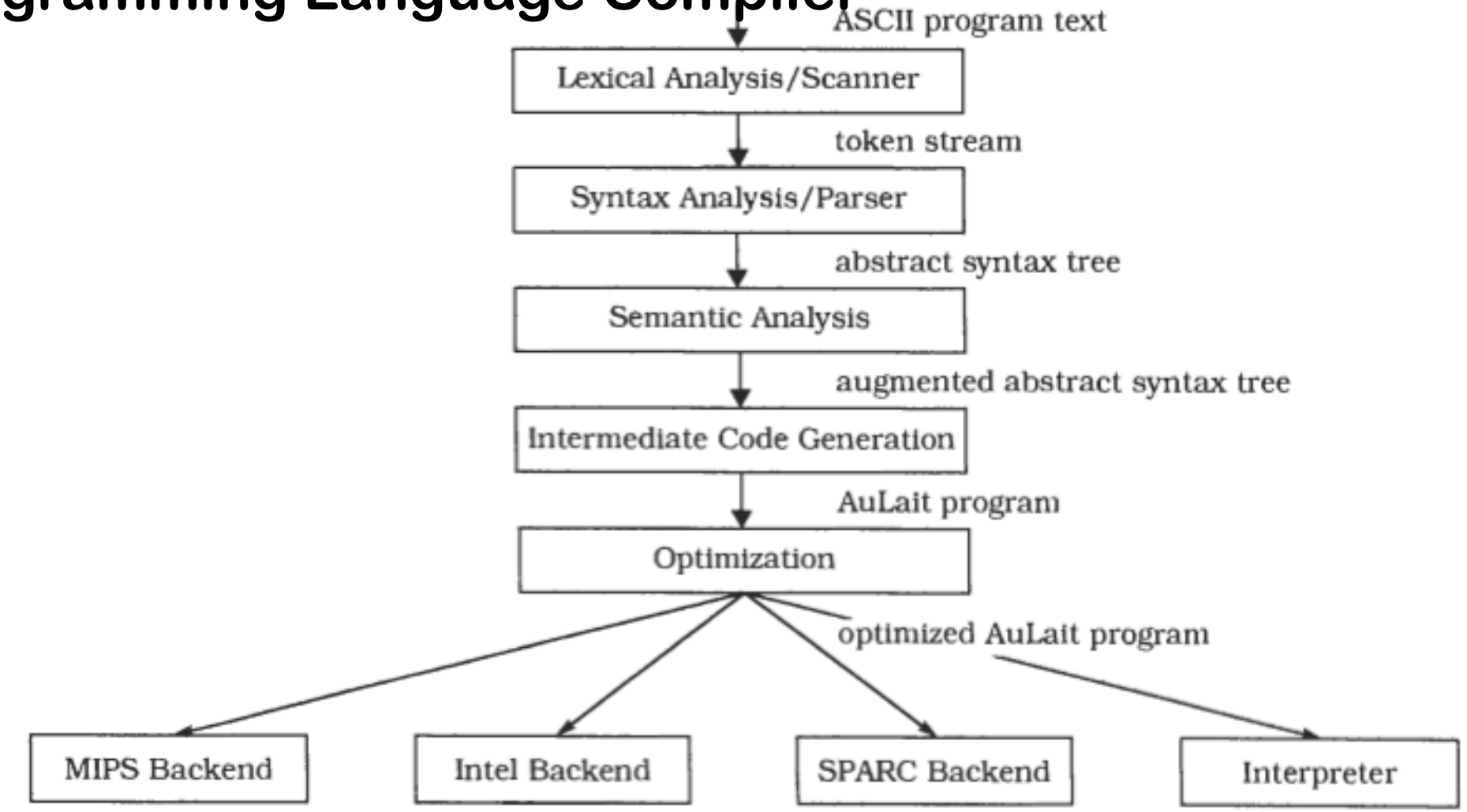
# Pipe-and-Filter Pattern: Definition

- 정의
  - 데이터 스트림을 처리하는 패턴
  - 데이터는 **Pipe**를 통해서 **Filter**로 전달
  - 전달된 데이터는 **Filter**를 통해 걸러지고 **pipe**를 통해 다음 **Filter**로 이동
- 예제: **Unix command system**
  - **cat etc/passwd | grep “joe” | sort > junk**



# Pipe-and-Filter Pattern: Example

- Programming Language Compiler



# Pipe-and-Filter Pattern: Description

- 정황 (**Context**)
  - 입력된 데이터 스트림을 처리하거나 변환해야 함
- 문제 (**Problem**)
  - 다수의 개발자가 참여하여 시스템을 구축해야 함
  - 데이터 스트림을 처리하는 프로세싱 단계가 쉽게 변할 수 있음
  - 프로세싱 단계를 재사용할 수 있어야 함
  - 다양한 방식으로 처리 결과물을 보여줄 수 있어야 함

# Pipe-and-Filter Pattern: Description

- 해법 (Solution)
  - 입력된 데이터를 처리하는 단계를 순차적(**sequential**)으로 처리
  - 단계들은 데이터 흐름에 의해 연결
    - 하나의 처리 단계(**Filter**로 구현)에서 처리된 결과물은 다음 처리단계의 입력물
  - **Filter**는 데이터를 증분(**incremental**)로 처리해 전달
    - 입력이 일정량 들어오는 대로 출력을 바로바로 산출
      - 낮은 지연시간(**Latency**)를 얻을 수 있음
      - 병렬 프로세싱을 가능하게 함
  - 각각의 처리 단계를 **Pipe**로 연결
    - **Pipe**는 인접한 **Filter**등 간의 데이터 흐름을 처리

# Pipe-and-Filter Pattern: Components

**Filter**

- 입력 데이터를 얻음
- 입력 데이터에 해당하는 함수를 수행
- 출력 데이터를 공급

**Pipe**

- 데이터를 전송
- 데이터를 버퍼링
- 인접한 능동 컴포넌트들을 동기화

**Data Source**

- 프로세싱 파이프라인에 입력을 전달

**Data Sink**

- 출력 결과물을 사용

# Pipe-and-Filter Pattern: Components

- **Filters**
  - 프로세싱 단위(**Processing unit**)의 역할
  - 데이터를 보강, 정제, 혹은 변형하는 프로세스를 진행
    - 보강(**enrich**): 정보를 계산하거나 추가
    - 정제(**refine**): 정보를 응축하거나 추출
    - 변형(**transform**): 정보를 다른 표현 형태로 가공
  - 데이터가 입력되면 시작
    - **Passive Filter: pull/push**하는 함수/프로시저의 호출을 받아 활성화됨
      - **Pull**: 후속(**subsequent**) **pipe**가 **Filter**로부터 출력 데이터를 끌어 옴
      - **Push**: 이전(**previous**) **pipe**가 **Filter**에 입력 데이터를 밀어 냄
    - **Active Filter: Thread** 등 자체 프로세싱 진행, 입력을 끌어오고 출력을 밀어냄

# Pipe-and-Filter Pattern: Components

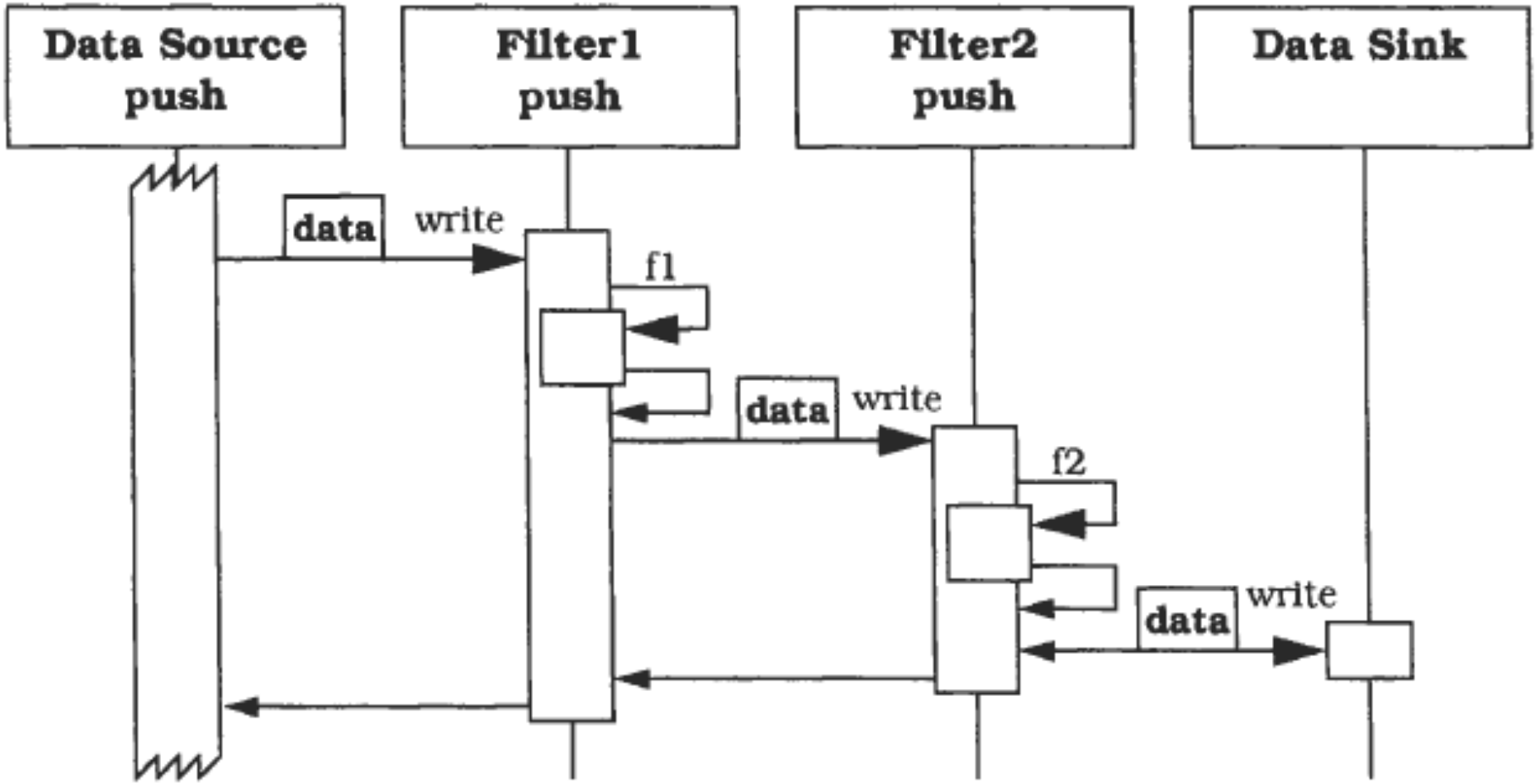
- **Pipes**
  - 연결된 **Filter**에게 데이터를 입력하고 출력하는 역할
  - 데이터가 흐르는 공간
    - 필터와 필터 사이를 연결
    - 데이터 소스와 최초 필터 간의 연결
    - 최종 필터와 데이터 싱크 간의 연결
  - 두 개의 **Active Filter** 연결 시
    - 이 두 개의 **Active Filter**를 동기화하는 역할 수행
      - **First-In-First-Out** 버퍼와 함께 동작

# Pipe-and-Filter Pattern: Components

- **Data Source**
  - 시스템에 들어오는 입력 값
  - 동일한 구조체나 타입으로 이뤄진 일련의 데이터 값을 제공
    - 예**1:** 여러 행의 텍스트로 이루어진 파일
    - 예**2:** 일련의 숫자를 감지하는 센서
- **Data Sink**
  - **Pipe**를 통해 나오는 최종 결과물을 취합
    - 예: 파일, 터미널, 애니메이션 프로그램

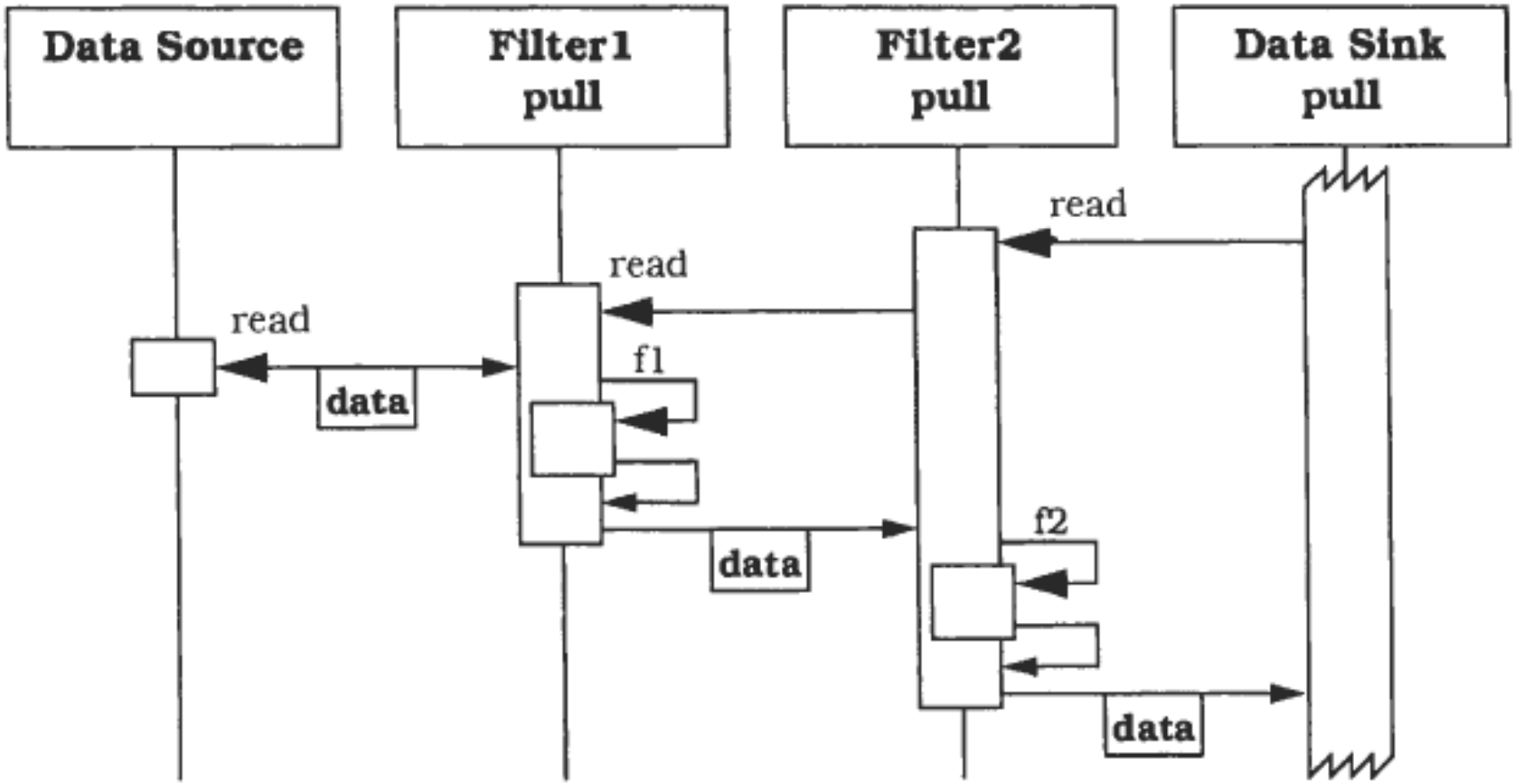
# Pipe-and-Filter Pattern: Behavior

- 시나리오 1. 데이터 소스에서 푸쉬 동작, 수동 필터에 데이터 작성



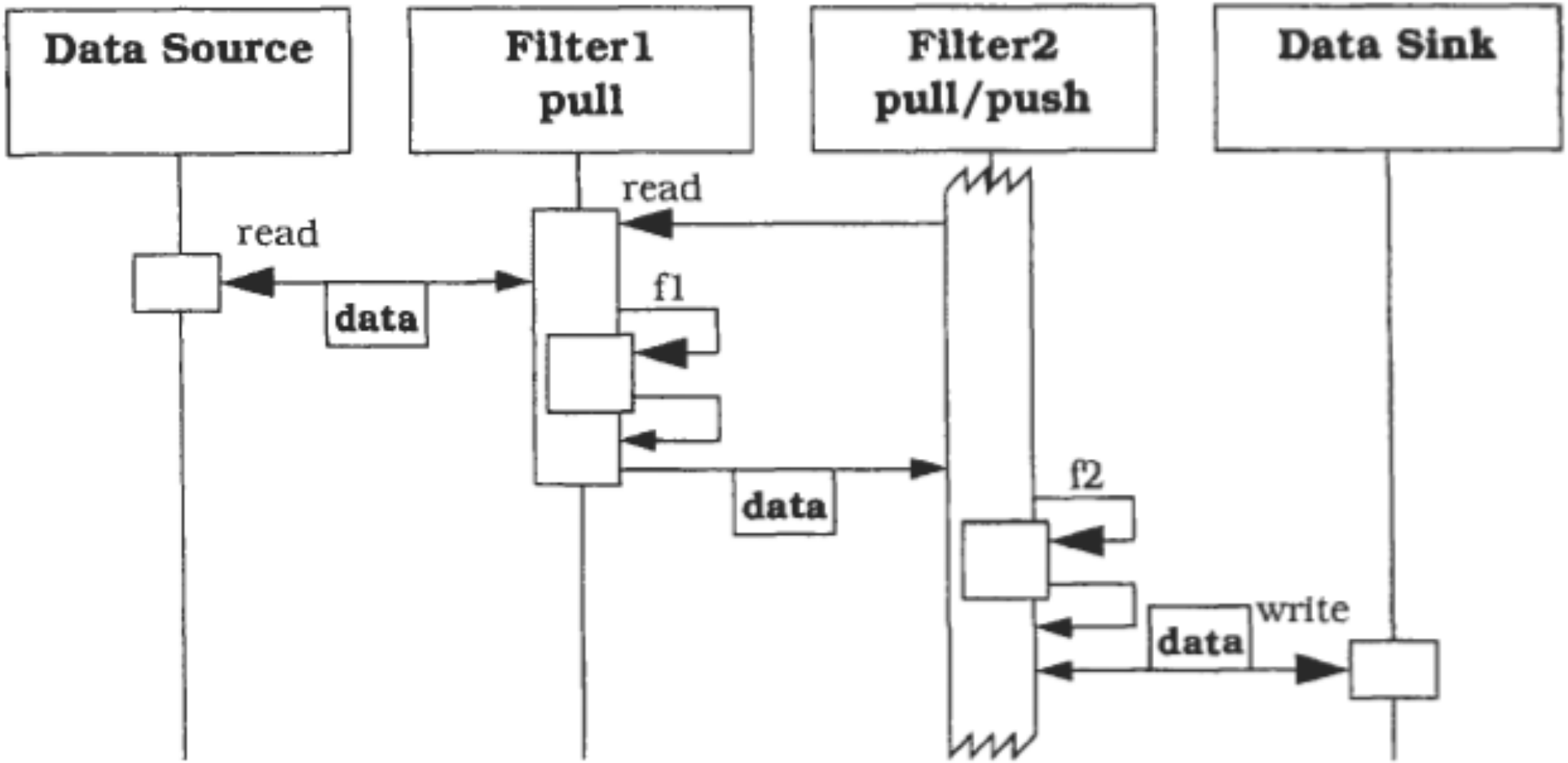
# Pipe-and-Filter Pattern: Behavior

- 시나리오 **II.** 데이터 싱크에서 풀**(pull)** 동작 작성



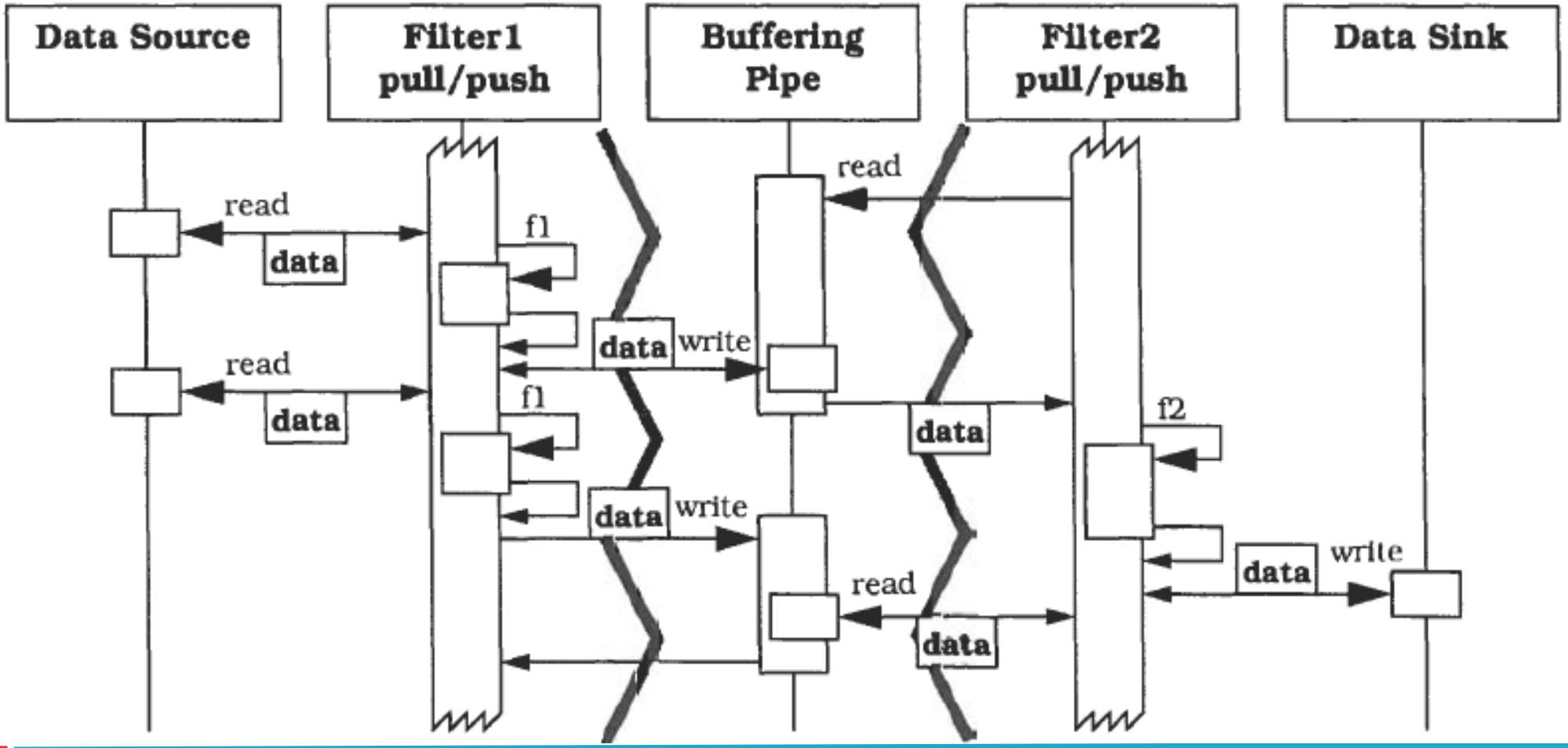
# Pipe-and-Filter Pattern: Behavior

- 시나리오 III. 두 번째 필터가 능동 역할 수행



# Pipe-and-Filter Pattern: Behavior

- 시나리오 **IV.** 모든 필터가 능동 필터, 동기화 파이프 존재



# Pipe-and-Filter Pattern: Behavior

- 시나리오 **IV.** 모든 필터가 능동 필터, 동기화 파이프 존재
  - **Filter2:** 파이프로부터 데이터를 읽기를 시도.
    - 버퍼가 비어 있어 데이터 요청은 **Filter2**를 중지시킴
  - **Filter1:** 데이터 소스로부터 데이터를 가져와 함수 **f1**실행
    - **Filter1**은 파이프로 결과를 밀어냄
  - **Filter2:** 데이터를 읽는 동작을 시작. **Filter1**은 버퍼가 찰 때까지 동작 진행
    - **Filter2**는 함수 **f2**를 계산하고 그 결과를 데이터 싱크에 작성

☞ **Filter1**과 **Filter2**는 능동 필터로 파이프의 버퍼 데이터로 읽고 쓰고 중단하는 작업 진행

# Pipe-and-Filter Pattern: Realization

- 구현
  - 파이프 연결: 메시지 큐나 유닉스 파이프와 같은 시스템 서비스 사용
  - 혹은 직접 호출 구현과 같은 방식 사용 가능
- 구현 순서
  1. 처리 단계 순서에 맞게 시스템의 작업을 나눔
     - 각 단계는 이전 단계의 아웃풋에만 의존성을 가져야 한다.
  2. 데이터 포맷을 결정
     - 단 하나의 포맷으로 정의하는 것이 가장 유연성이 높고 조합이 쉽다.
     - 혹은 데이터를 변환하기 위한 데이터 변환 필터 컴포넌트를 만들 수 있다.

# Pipe-and-Filter Pattern: Realization

- 구현 순서
  3. 파이프 간의 연결 방법을 설계한다.
     - 필터를 능동 컴포넌트 혹은 수동 컴포넌트로 구현하는지에 따라 결정
     - (수동 필터) 필터 컴포넌트 재조합을 위해서 항상 코드를 변경해야 함
     - (능동 필터) 독립적 파이프 메커니즘 사용, 파이프에서 **FIFO** 버퍼 제공
  4. 필터 설계하고 구현한다.
     - 풀 동작을 위한 함수, 푸시 동작을 위한 함수를 수동 필터 구현 가능
     - 프로세스 혹은 스레드 역할을 하도록 능동 필터 구현 가능
  5. 에러 핸들링을 설계하고 구현한다.
  6. 파이프라인을 구현한다.

# Pipe-and-Filter Pattern: Implementation

- Filter Code

```
public abstract class Filter<I, O> extends ThreadedRunner {
        protected Pipe<I> input;
        protected Pipe<O> output;

        public Filter(Pipe<I> input, Pipe<O> output) {
                this.input = input;
                this.output = output;
        }

        @Override
        public void run() {
                transformBetween(input, output);
        }

        protected abstract void transformBetween(Pipe<I> input, Pipe<O> output);
}
```

# Pipe-and-Filter Pattern: Implementation

- Pipe Code

```
public interface Pipe<T> {
    public boolean put(T obj);
    public T nextOrNullIfEmptied() throws InterruptedException;
    public void closeForWriting();
}
```

# Pipe-and-Filter Pattern: Implementation

- DataSource Code

```
public abstract class Generator<T> extends ThreadedRunner {
   protected Pipe<T> output;

   public Generator(Pipe<T> output) {
      this.output = output;
   }

   @Override
   public void run() {
      generateInto(output);
   }

   public abstract void generateInto(Pipe<T> pipe);
}
```

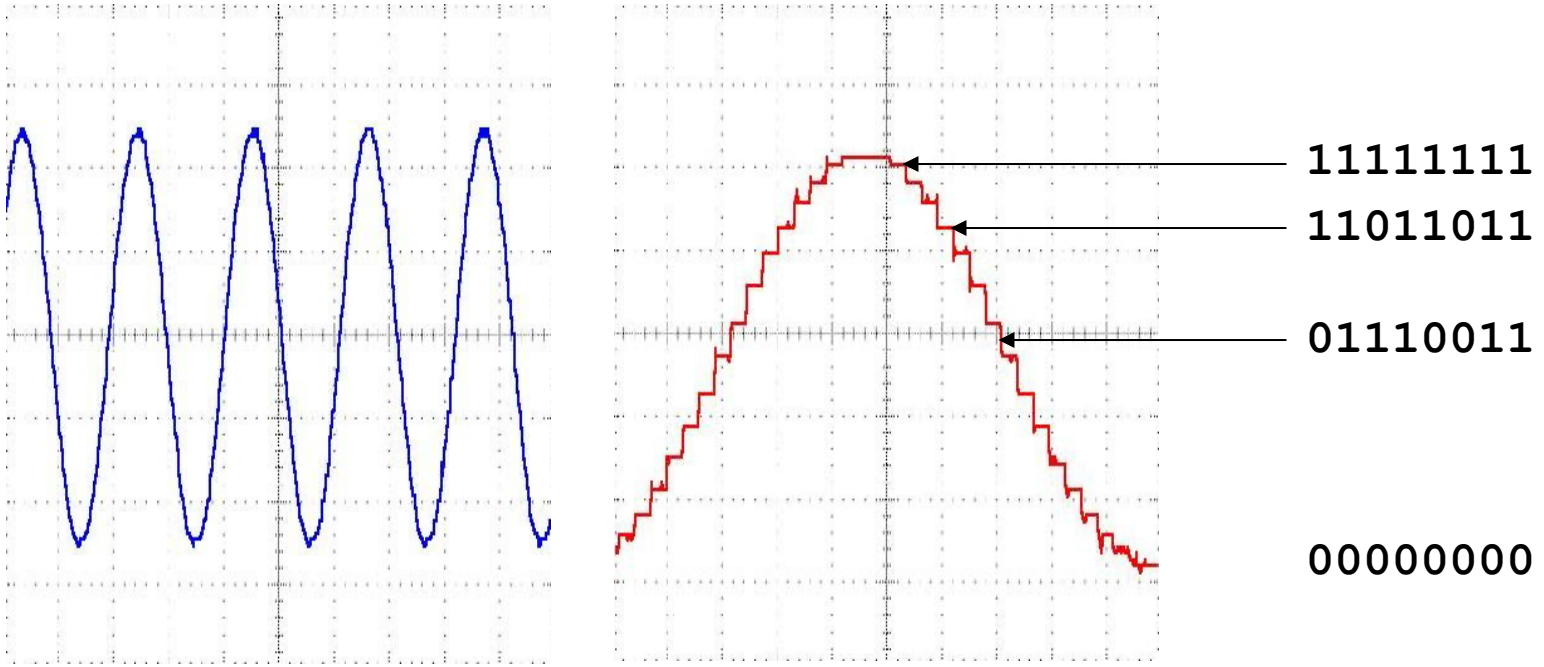
# Pipe-and-Filter Pattern: Implementation

- DataSink Code

```
public abstract class Sink<T> extends ThreadedRunner {
   protected Pipe<T> input;

   public Sink(Pipe<T> input) {
      this.input = input;
   }

   @Override
   public void run() {
      takeFrom(input);
   }

   public abstract void takeFrom(Pipe<T> pipe);
}
```

# Pipe-and-Filter Pattern: Case Studies

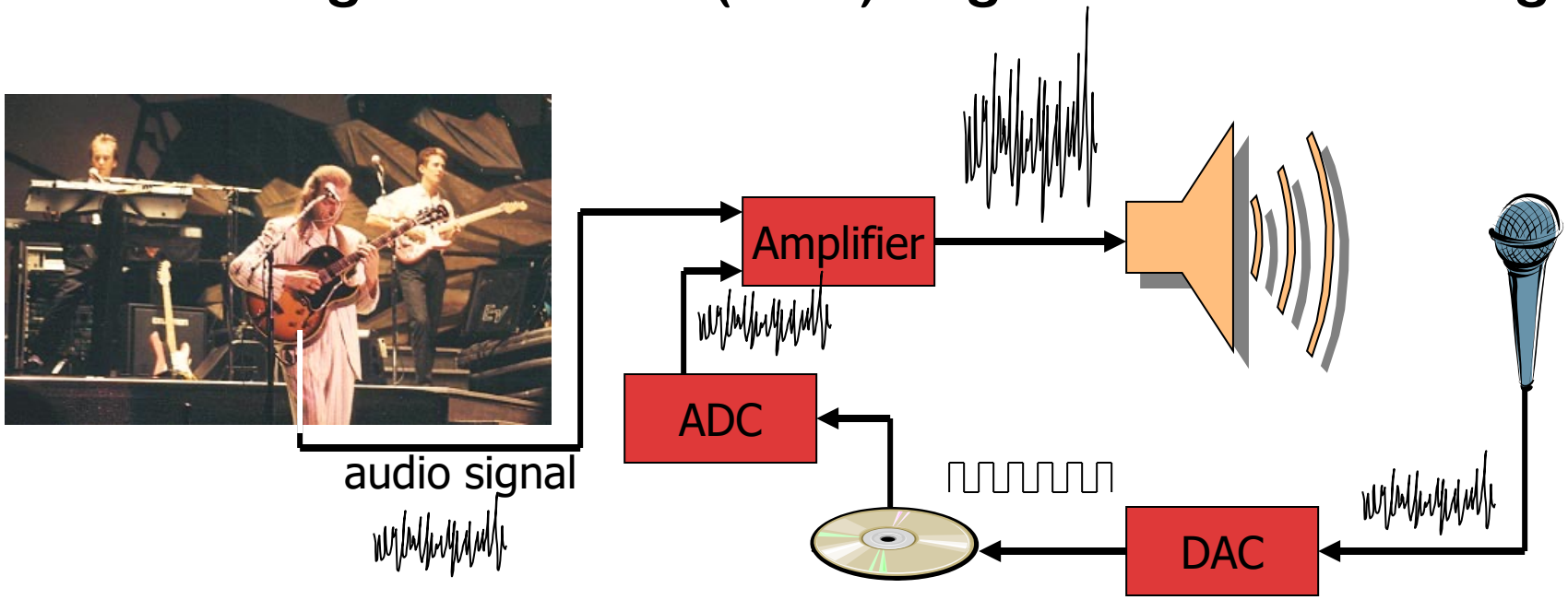
- 시그널 처리와 관련된 어플리케이션에 주로 사용
  - radar
  - medical
  - process control
  - sonar
  - audio
  - video
  - telemetry 등등




Signaling Process: 시그널 처리는 지속적으로 들어오는 아날로그 신호를 디지털 신호로 바꿔주고 다시 디지털 신호를 아날로그 신호로 바꿔주는 방식으로 개발

# Pipe-and-Filter Pattern: Case Studies

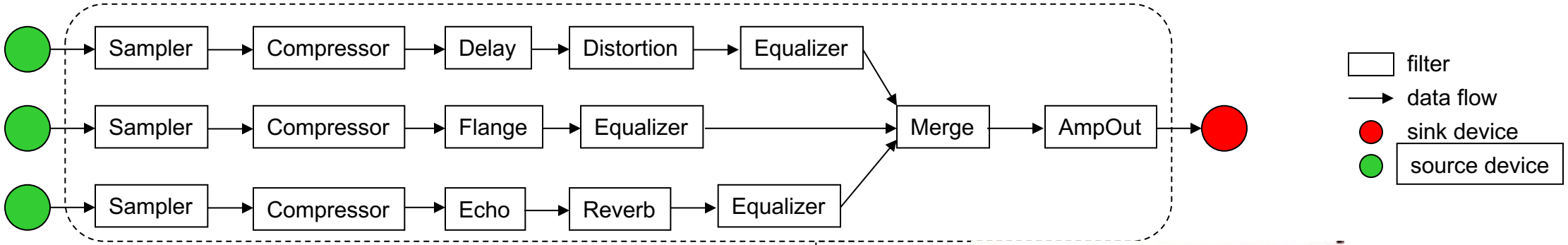
- Digital Audio
  - Analog to Digital Converters (ADC): analog voltages to digital values.
  - Digital to Analog Converters (DAC): digital values to voltages.



# Pipe-and-Filter Pattern: Case Studies

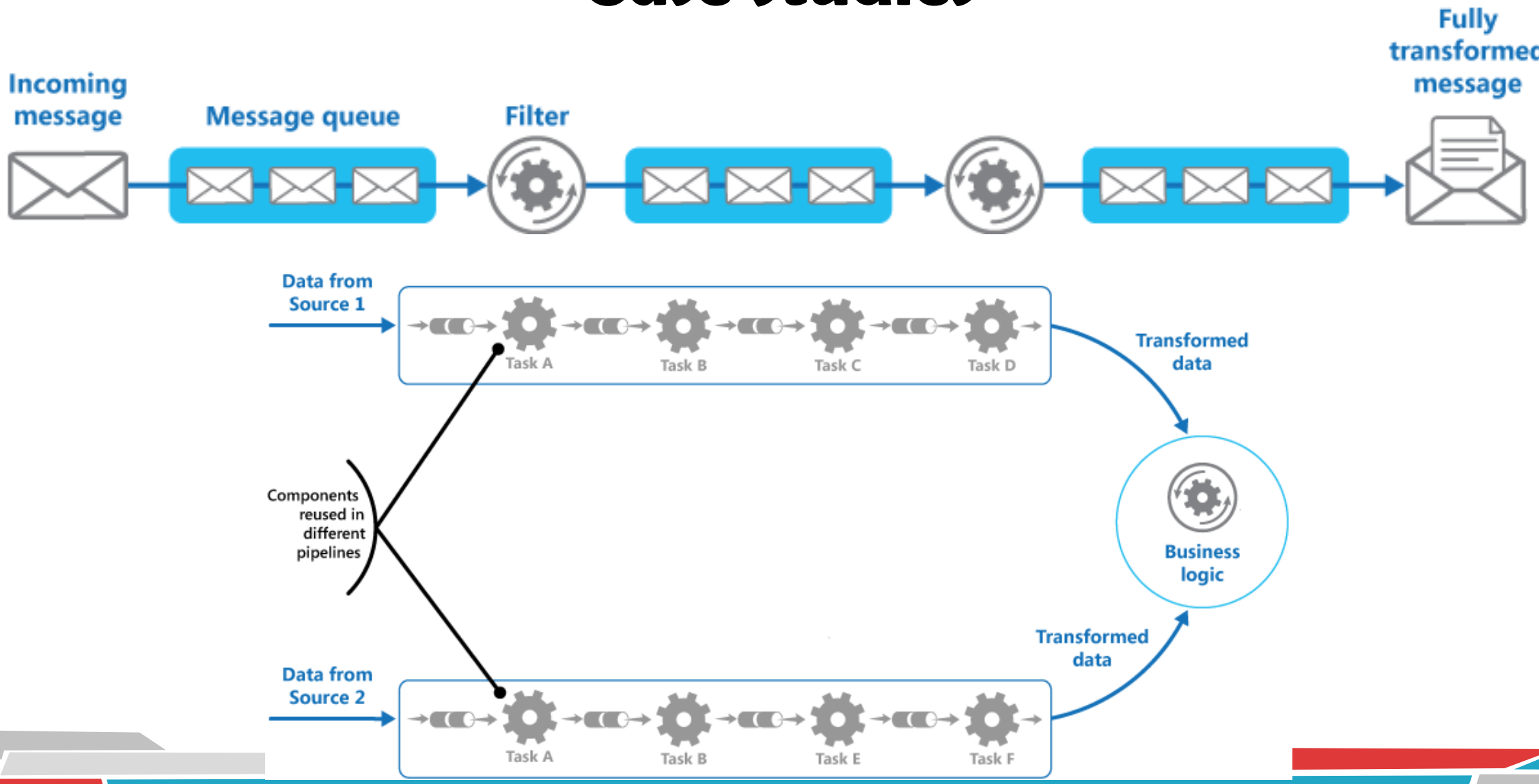
## Modern Audio Application: Digital Recording

Note: this is an oversimplification and there are many other ways to set up a digital recording system



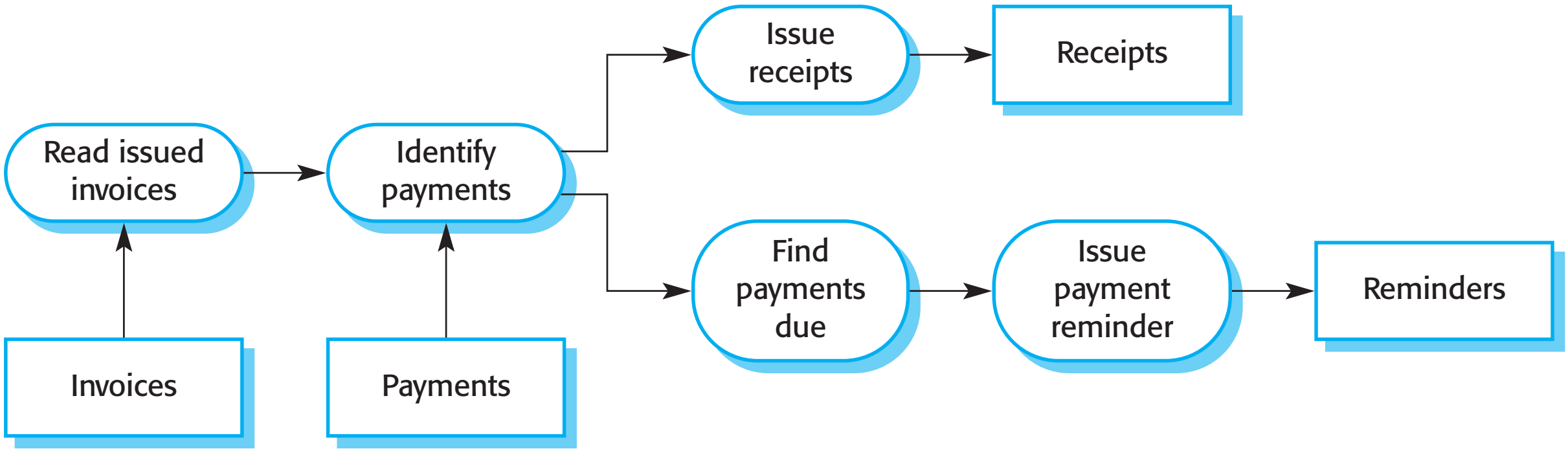

All in done in software…

# Pipe-and-Filter Pattern: Case Studies




https://docs.microsoft.com/ko-kr/azure/architecture/patterns/pipes-and-filters

# Pipe-and-Filter Pattern: Case Studies



# Pipe-and-Filter Pattern: Benefits

- 중간 결과 파일이 불필요함
  - 중간 결과 파일을 사용할 경우: 오류 확률 높임, 병렬 계산 어렵게 함
  - 해당 패턴 사용 시 파이프라인에 접합하여 중간 데이터 검사 가능
- **Filter**의 교환이나 재조합이 쉬움
  - 필터를 쉽게 교환할 수 있도록 인터페이스가 간단 (단, 런타임 필터 교환 불가)
- **Filter**의 재사용성이 좋음
  - **Active Filter**인 경우 재사용성 높아짐; 셸 사용 시 엔드 유저 환경 지원
- **Parallel** 프로세싱에 용이함
  - **Active Filter**를 병렬로 개시하는 것이 가능

# Pipe-and-Filter Pattern: Liabilities

- 병렬 프로세싱을 사용해 효율성을 높이는 부분이 힘들 수 있음; 다음 요소가 성능에 악영향
  - **Filter** 간의 데이터 전송 시간 및 비용
  - 다중 입력을 받는 **Filter**의 경우 모든 입력을 받아야 출력을 산출
  - **Thread**나 **Process**간의 **Context switching** 시간
  - **Synchronization**으로 인하여 **Filter**를 중지했다가 시작하는 시간
- 데이터 포맷 변환에 과부하 발생 가능성 있음
  - 필터의 입출력에 단 하나의 데이터 타입을 사용하는 경우
- 에러 처리가 어려움 (데이터 복구 작업 등)

# Question?

Seonah Lee
saleese@gmail.com