소프트웨어 아키텍처 설계 패턴

김정호 대표이사

패턴 개요

- 패턴
  - 특정 문제에 대한 해법을 추상화하고 그 안의 공통된 요인을 추출하여 정형화한 것을 패턴이라고 한다.
- 패턴의 3가지의 스키마
  - 정황(Context) : 문제를 발생시키는 상황
  - 문제(Problem) : 해당 정황에서 반복적으로 발생하는 문제
  - 해법(Solution) : 해당 문제에 대해 검증된 해답
- 소프트웨어 시스템 개발에서의 패턴 종류
  - 소프트웨어 아키텍처 패턴
    - 문제를 해결하는 해법으로 소프트웨어 시스템의 기본 구조와 관련된 것을 다룰 경우 아키텍처 수준의 패턴이라고 말한다.
  - 디자인 패턴
    - 소프트웨어 시스템의 서브시스템이나 컴포넌트들, 혹은 그것들 간의 관계를 해법으로 사용하는 경우, 디자인 패턴이라고 말한다.
  - 이디엄 (Idiom)
    - 특정 프로그래밍 언어의 기능을 이용하여 컴포넌트들 혹은 컴포넌트들 간 관계의 특정 측면을 구현하는 방법을 이디엄이라고 한다.

아키텍처 패턴 종류

- Layer
- Blackboard
- Pipes and Filters
- Broker
- Model-View-Controller
- Publisher-Subscriber

- Presentation-Abstraction-Control
- Microkernel
- Reflection
- Interceptor
- Reactor
- Procator
- Half-sync/half-async
- Leader/Followers

Layer 패턴

❖ 정의

- 특정 추상 레벨에 있는 서브태스크들끼리 서로 묶어서 하나의 그룹으로 분류하는 방식
- 하위 수준의 이슈를 상위 수준에 이슈와 분리시켜 소프트웨어의 재사용성을 높여주는 패턴

❖ 예제

- 네트워크 프로토콜 아키텍처 (e.g. OSI 7 layer)
- 가상 머신 (e.g. interpreters, JVM)

Layer 패턴

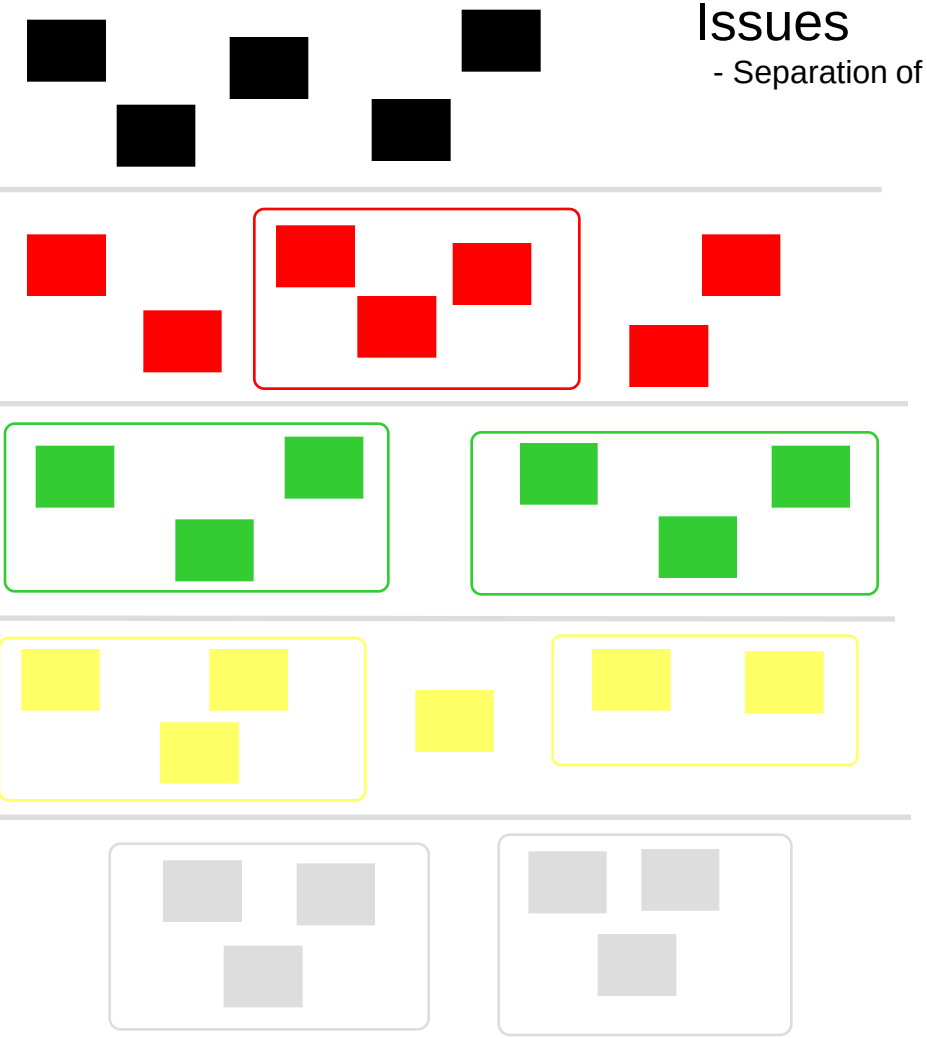
Applications& interfaces

Major processes

Domain classes

Mechanisms

Services

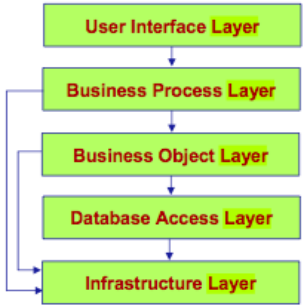


Issues
- Separation of concerns

Layer 패턴

❖ 정황(Context)

- 시스템의 규모가 커서 분해할 필요가 있을 경우



❖ 문제(Problem)

- 하위 레벨과 상위 레벨 이슈가 서로 혼재해 있다는 점이 주된 특징인 시스템을 설계할 경우
- 시스템의 기능이 수직적으로 나눠져 있거나 수평적인 경우와 혼재되어 있는 경우

❖ 해법(Solution)

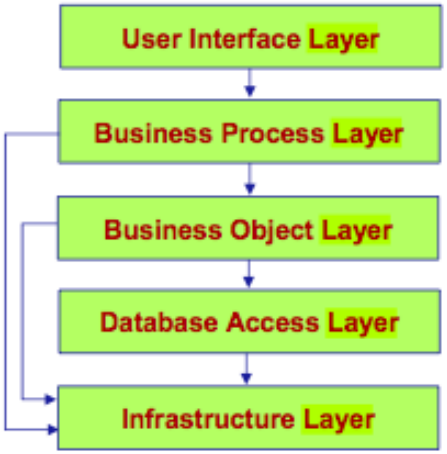
- 시스템의 상호연동 관계가 있는 모듈들을 모아 계층으로 추상화 (최하위 계층: Layer 1, 최상위 계층 : Layer N)
- Layer J는 반드시 Layer J-1이 제공하는 서비스만 사용. 다른 계층의 서비스를 사용해서는 안됨

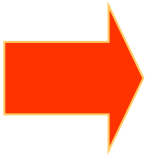
Layer 패턴

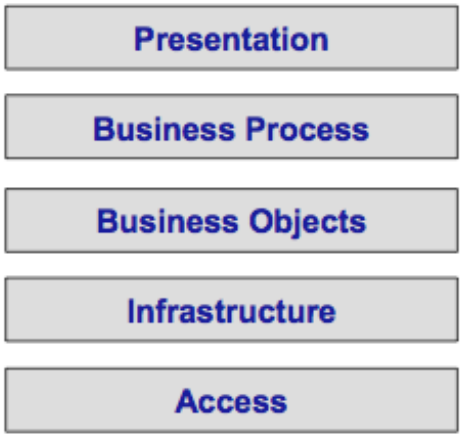
❖ 설계 순서

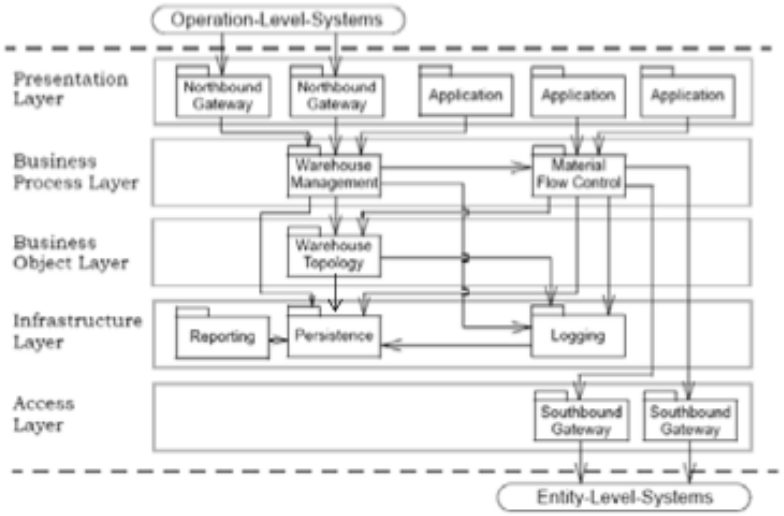
1. 계층 별로 모듈을 묶는 추상 기준을 정의
2. 추상 기준에 따라 계층을 몇 레벨로 나눌지 결정
3. 계층마다 역할 및 태스크 부여
4. 계층별 제공 서비스를 상세히 정의
5. 계층별 상세 인터페이스 정의
6. 시스템 기능이 계층에서 동작하는 것이 가능한지 확인
   (예. 유스케이스 시나리오를 시뮬레이션 하는 방식)
7. 계층 내부에 대한 구조 정의
8. 인접한 계층 간의 통신 방식 정의
9. 예외 처리 방식을 정의

Layer 패턴

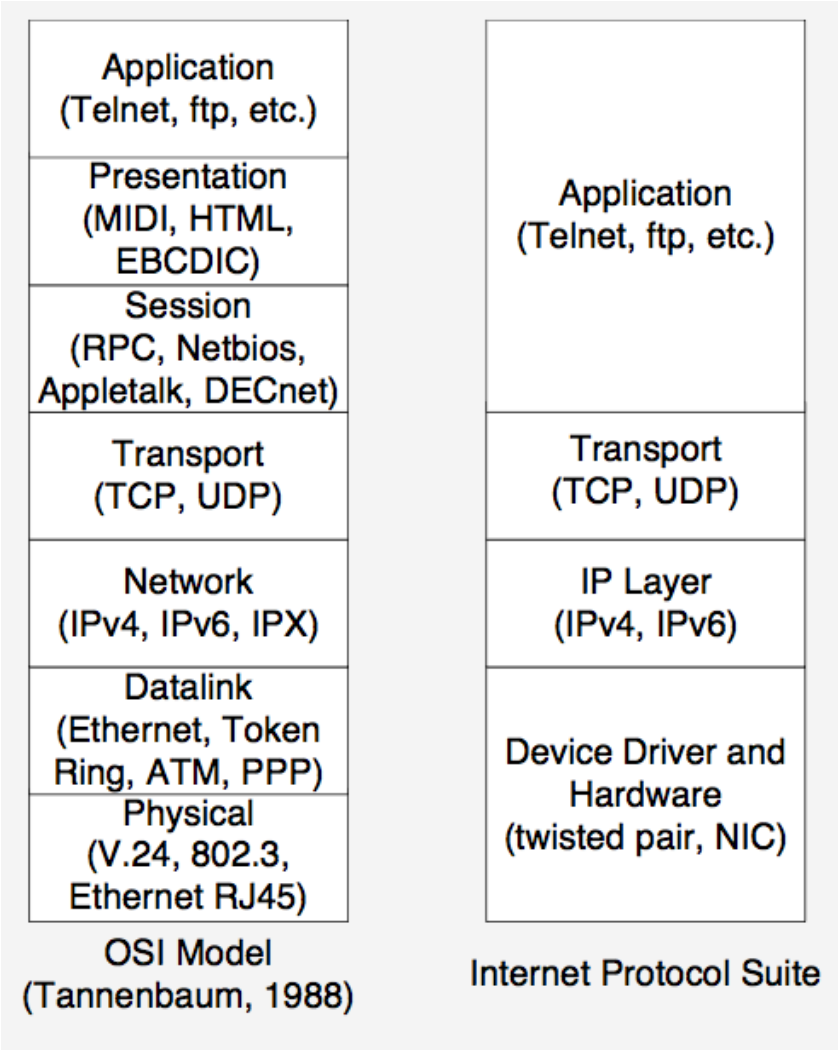

User Interface Layer
Business Process Layer
Business Object Layer
Database Access Layer
Infrastructure Layer



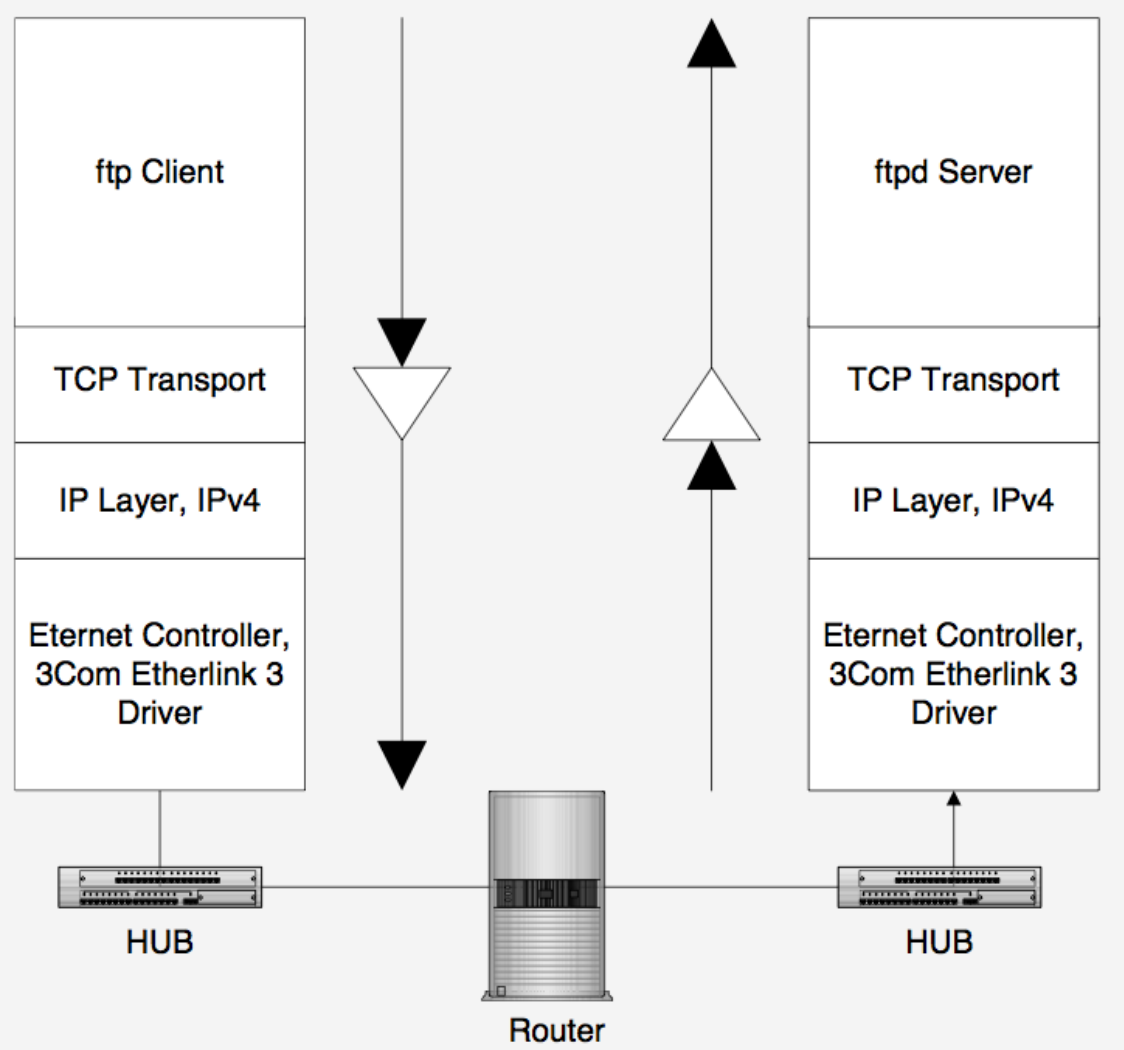

Presentation
Business Process
Business Objects
Infrastructure
Access


Operation-Level-Systems
Presentation Layer
Northbound Gateway
Northbound Gateway
Application
Application
Application
Business Process Layer
Warehouse Management
Material Flow Control
Business Object Layer
Warehouse Topology
Infrastructure Layer
Reporting
Persistence
Logging
Access Layer
Southbound Gateway
Southbound Gateway
Entity-Level-Systems

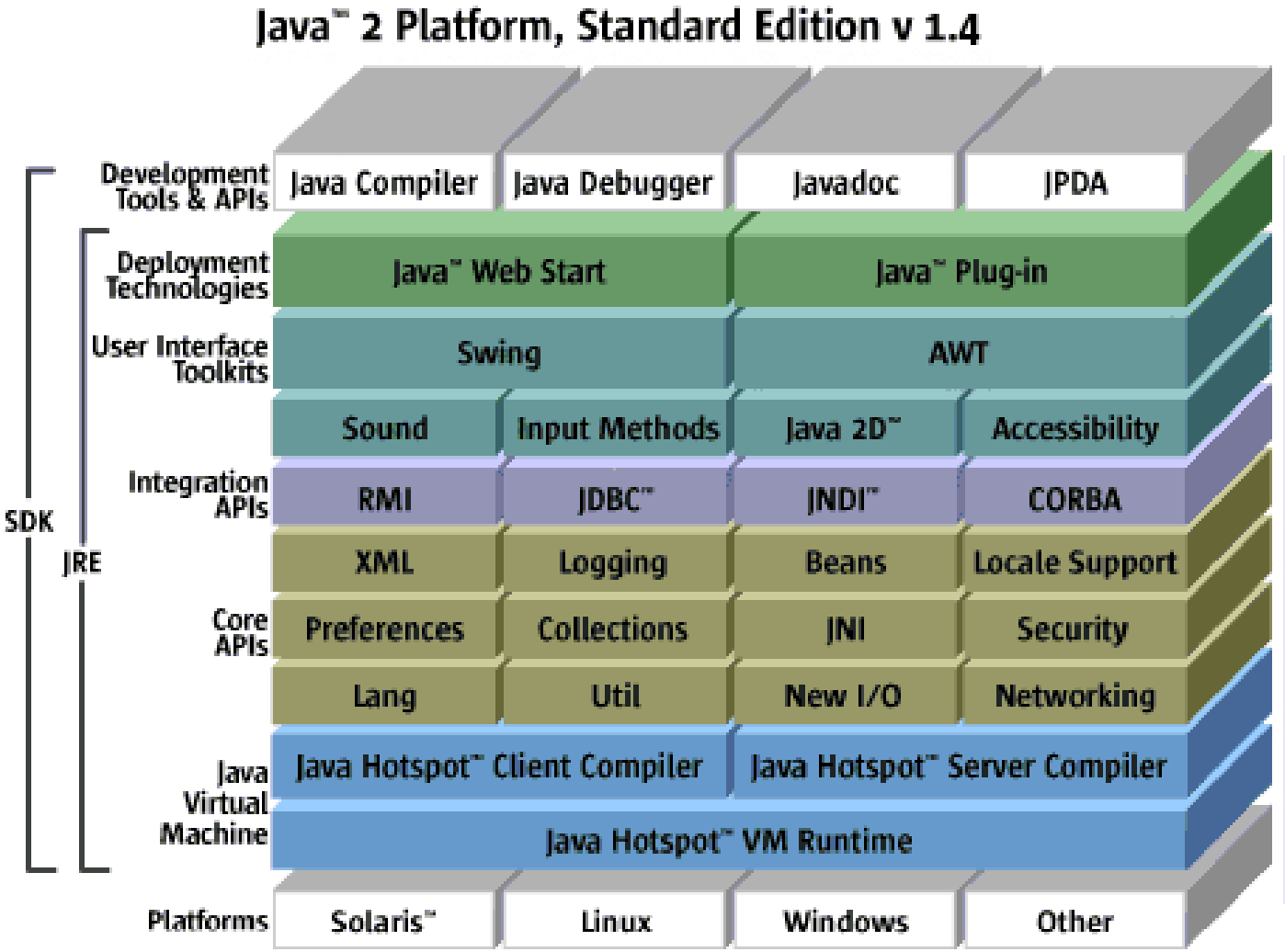
Layer 패턴


Application
(Telnet, ftp, etc.)
Presentation
(MIDI, HTML,
EBCDIC)
Session
(RPC, Netbios,
Appletalk, DECnet)
Transport
(TCP, UDP)
Network
(IPv4, IPv6, IPX)
Datalink
(Ethernet, Token
Ring, ATM, PPP)
Physical
(V.24, 802.3,
Ethernet RJ45)
OSI Model
(Tannenbaum, 1988)
Application
(Telnet, ftp, etc.)
Transport
(TCP, UDP)
IP Layer
(IPv4, IPv6)
Device Driver and
Hardware
(twisted pair, NIC)
Internet Protocol Suite

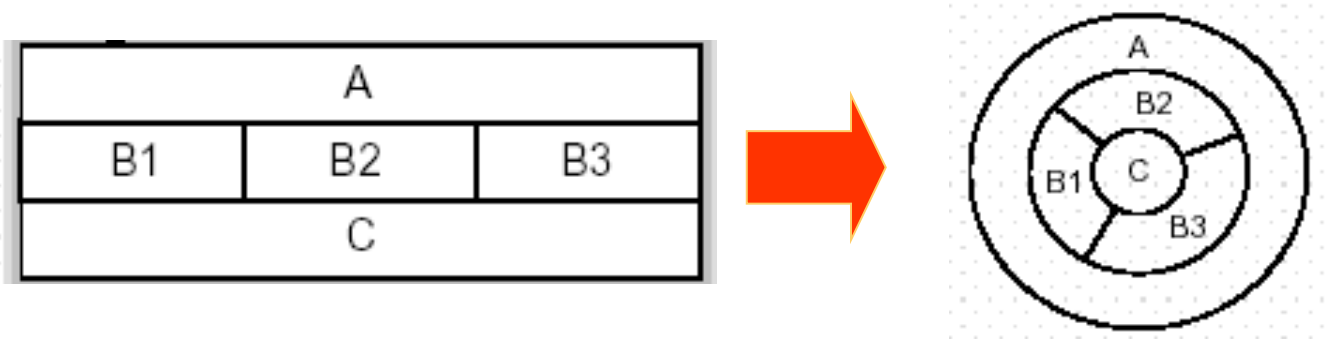
Layer 패턴


ftp Client
TCP Transport
IP Layer, IPv4
Eternet Controller, 3Com Etherlink 3 Driver
ftpd Server
TCP Transport
IP Layer, IPv4
Eternet Controller, 3Com Etherlink 3 Driver
HUB
HUB
Router

Layer 패턴


Java™ 2 Platform, Standard Edition v 1.4
Development Tools & APIs
Java Compiler
Java Debugger
Javadoc
JPDA
Deployment Technologies
Java™ Web Start
Java™ Plug-in
User Interface Toolkits
Swing
AWT
Sound
Input Methods
Java 2D™
Accessibility
Integration APIs
RMI
JDBC™
JNDI™
CORBA
XML
Logging
Beans
Locale Support
Core APIs
Preferences
Collections
JNI
Security
Lang
Util
New I/O
Networking
Java Virtual Machine
Java Hotspot™ Client Compiler
Java Hotspot™ Server Compiler
Java Hotspot™ VM Runtime
Platforms
Solaris™
Linux
Windows
Other
SDK
JRE

Layer 패턴



Layer 패턴

❖ 장점

- 계층별 연동을 한정할 수 있어 Loosely coupled 원칙을 지킬 수 있음
- 계층별로 변화에 대한 영향력을 한정할 수 있어 코딩이나 테스트를 계층별로 진행할 수 있음
- 인터페이스 정의가 잘 되어 있다면 계층을 통째로 교체할 수 있음
- 모듈의 재사용성을 높여 유지보수성이나 이식성이 필요한 시스템에 적용하기 좋은 패턴임

❖ 단점

- 계층의 원칙을 지키기 위해 각 계층을 모두 거쳐야 하므로 성능 측면에 불이익을 받을 수 있음
- 계층을 구분하기 어렵고 잘못 구분할 경우 설계 수정이 빈번히 발생할 수 있음
- 계층의 적절한 개수 및 규모를 정의하는 것이 어려움

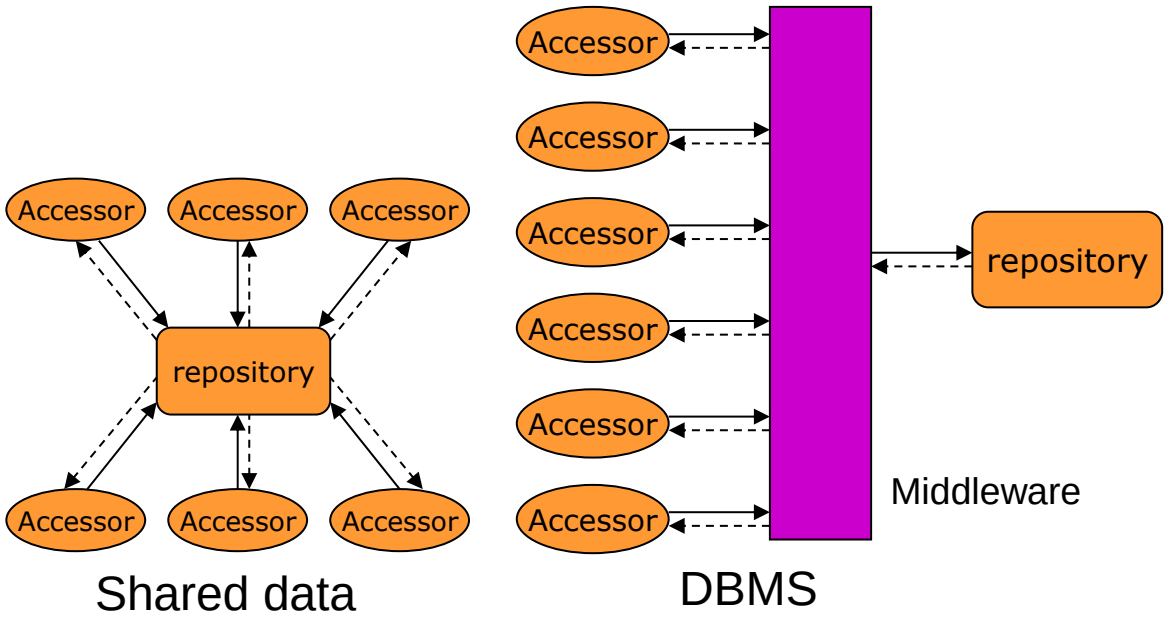
Blackboard 패턴

❖ 정의

- Shared data, database와 같은 데이터 중심 패턴 중에 하나
- 명확히 정의된 문제 해법이 없을 때 문제를 풀어가는 하나의 방식을 정의한 패턴
- 대략적으로 해법을 수립하기 위해 특수한 서비스 시스템의 지식을 조합하는 패턴

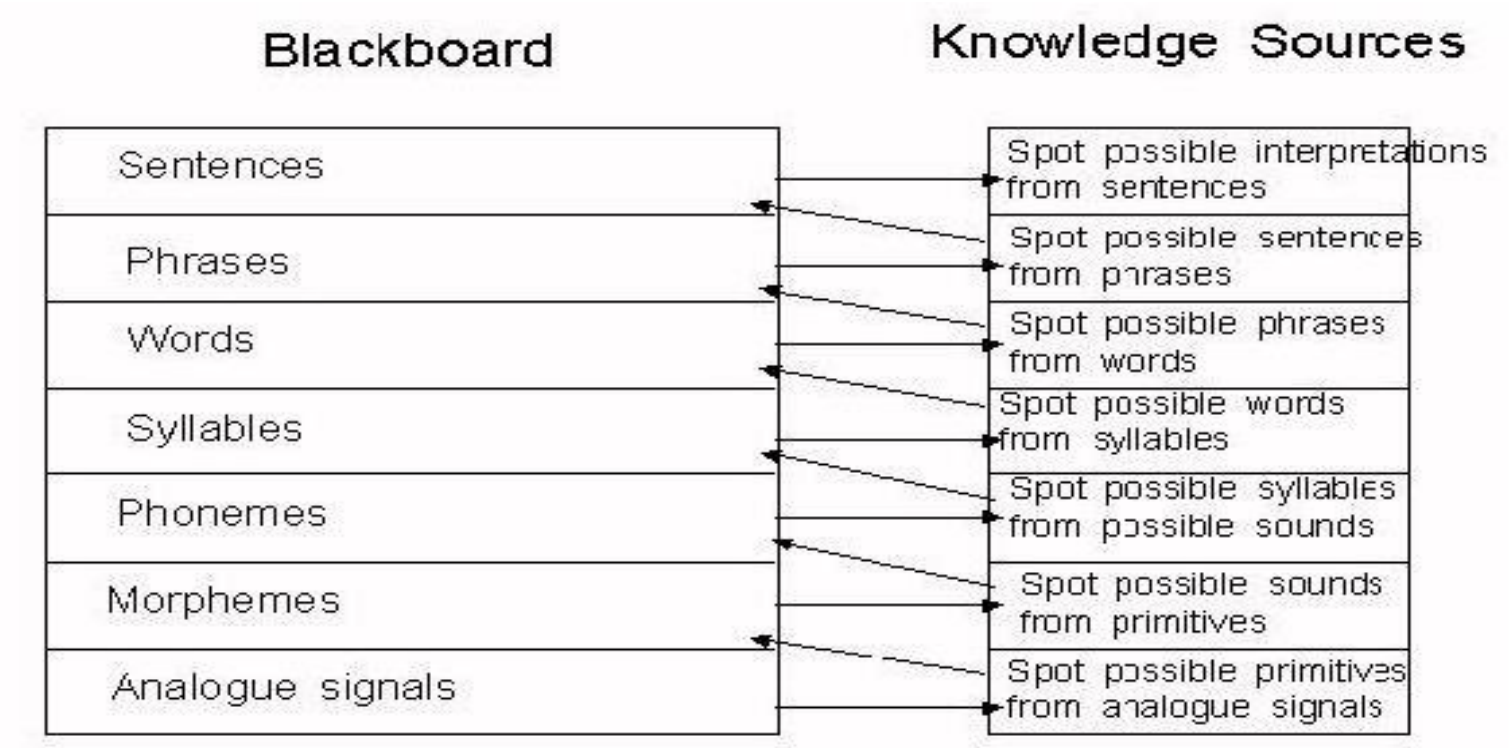
❖ 예제

- AI system
- Signal processing
- Radar/sonar
- Vision processing
- Speech processing



Blackboard 패턴

- 음성 인식 시스템
  - Input: 파형 형태의 음성
  - Output: 시스템이 인식한 문장



Blackboard 패턴

❖ 정황(Context)

- 도메인의 해법이 명확하지 않은 경우

❖ 문제(Problem)

- 컴퓨터 비전, 음성 인식, 화상 인식 등 도메인 분야와 같이 상위 수준의 데이터 구조로 변환하기 위한 명확한 해법이 존재하지 않음
- 하나의 문제를 분해하면 여러 가지 도메인의 하위 문제들이 발생함.
- 하위 문제들을 해법이 있다 해도 다시 취합하여 상위 수준의 문제를 포괄적으로 해결하는데 어려움이 있음

❖ 해법(Solution)

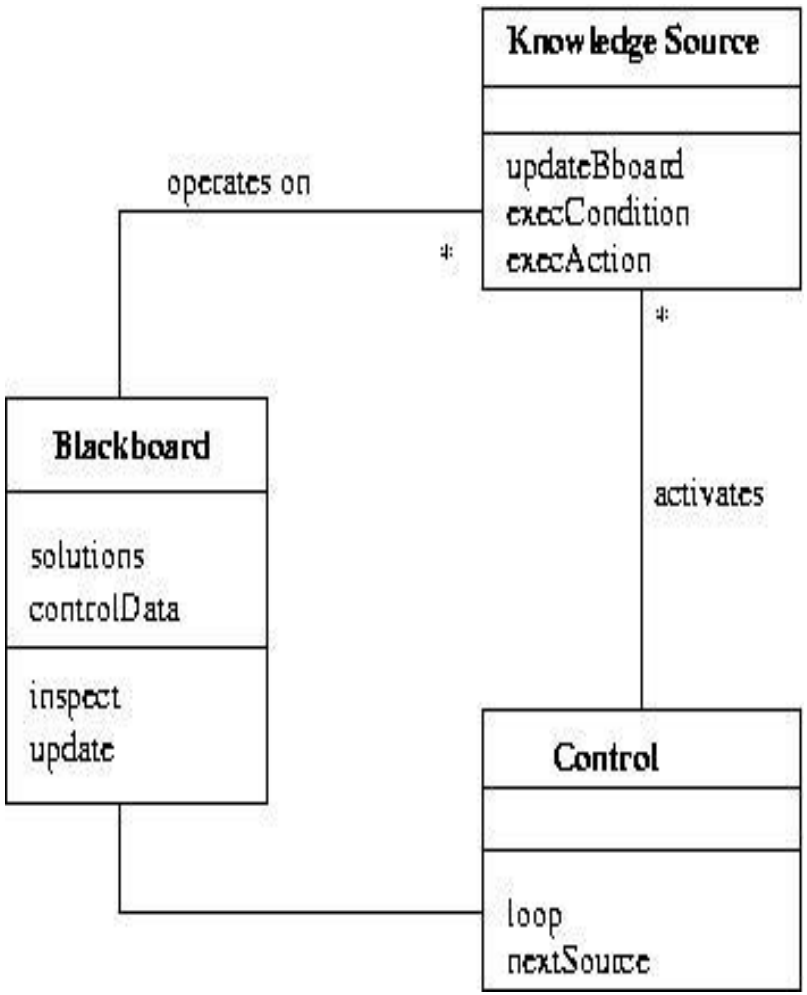
- 일반적인 데이터 구조(blackboard)를 가지고 각각 독립적으로 동작하는 프로그램들 (Knowledge sources)로 이뤄져 있음
- 통제 컴포넌트 (Control)에 의해 모든 독립적인 프로그램들은 하나의 해법을 찾기 위해 서로 협력하여 동작함

Blackboard 패턴

❖ 구성 컴포넌트

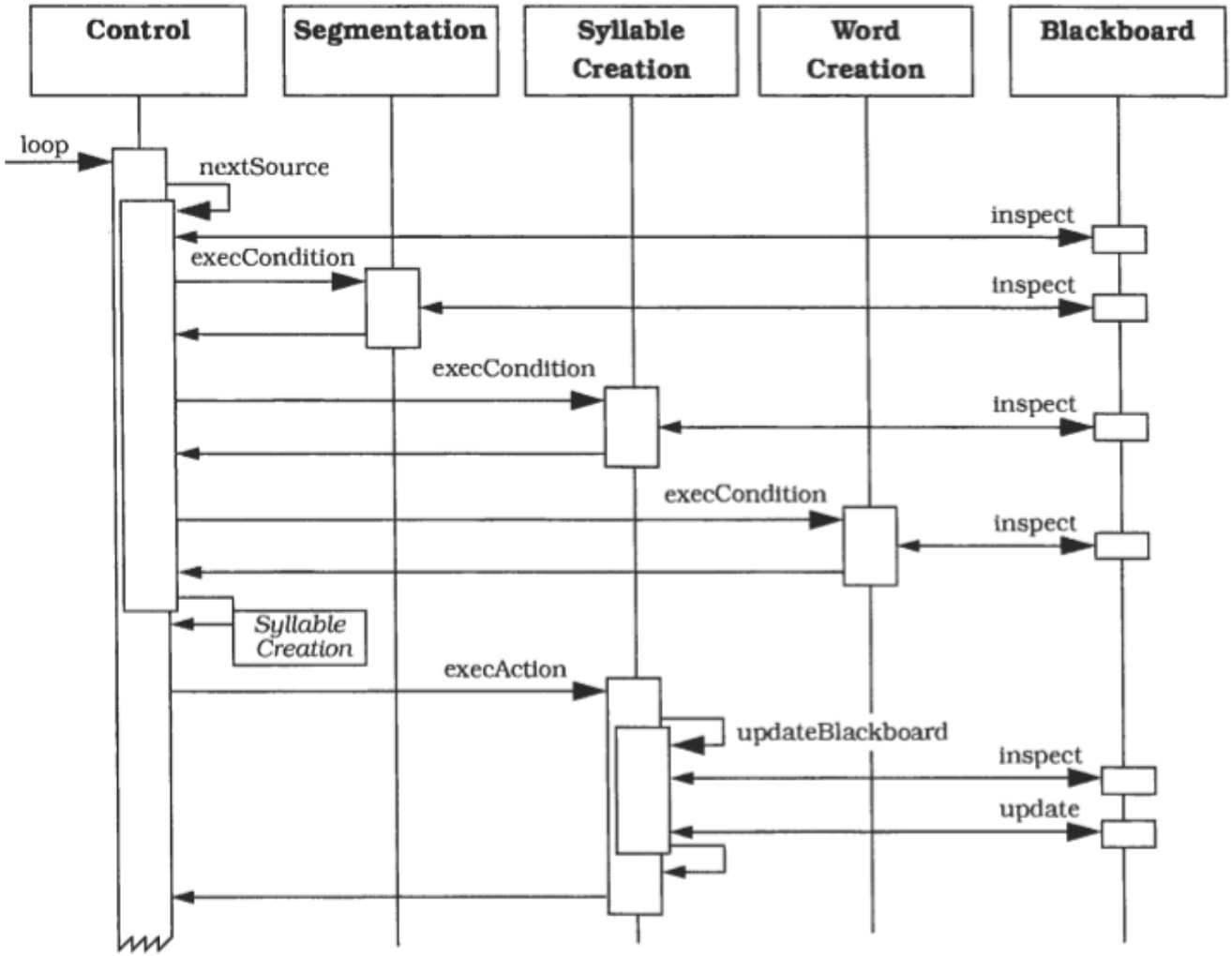
- Blackboard
  - 문제의 현재 상태를 제시
  - 데이터를 관리
- Knowledge Sources
  - Blackboard 상태를 업데이트
  - 특정 도메인의 해법을 제시
  - Blackboard에 그 해법을 적용
- Control
  - Blackboard 상태 모니터링
  - Knowledge Sources 스케줄을 관리하고 실행시킴

Blackboard 패턴



1. Start Control::loop
2. Control::nextSource
3. determine potential knowledge sources by calling Blackboard::inspect
4. Invoke KnowledgeSource::execCondition of each candidate knowledge source
5. Each candidate knowledge source invokes Blackboard::inspect to determine if/how it can contribute to current state of solution
6. Control chooses a knowledge source to invoke by calling KnowledgeSource::execAction
7. Executes KnowledgeSource::updateBlackboard
8. Calls Blackboard::inspect
9. Calls Blackboard::update

Blackboard 패턴

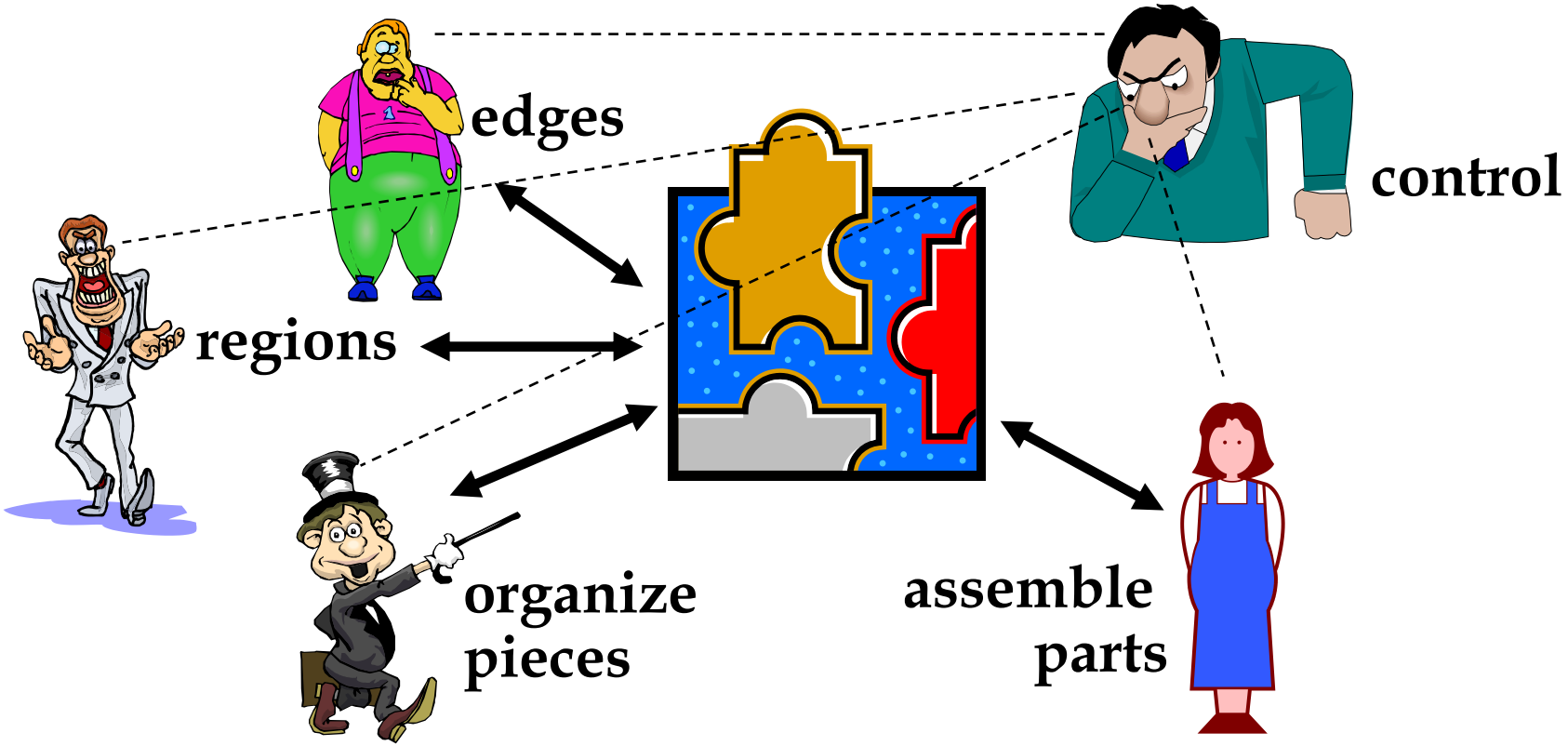


Blackboard 패턴

❖ 설계 순서

1. 문제의 도메인을 정의하고 해법을 찾기 위해 일반적인 지식 분야를 상세히 살펴본다.
2. 해법에 대한 추상화 수준을 상위 수준에서 하위 수준까지 나눠서 정의한다.
3. 해법 수준에 맞게 Knowledge source를 정이하고 각 수준으로 분할한다.
4. 모든 Knowledge source가 blackboard와 상호작용하는 표현 방식을 찾아서 정의한다. (blackboard 어휘를 정의한다.)
5. Control을 정의한다.

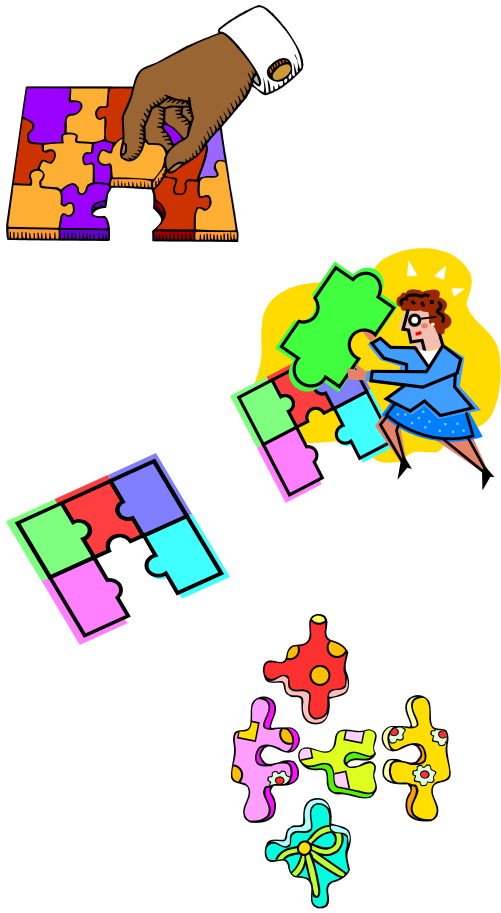
How do you solve a puzzle?



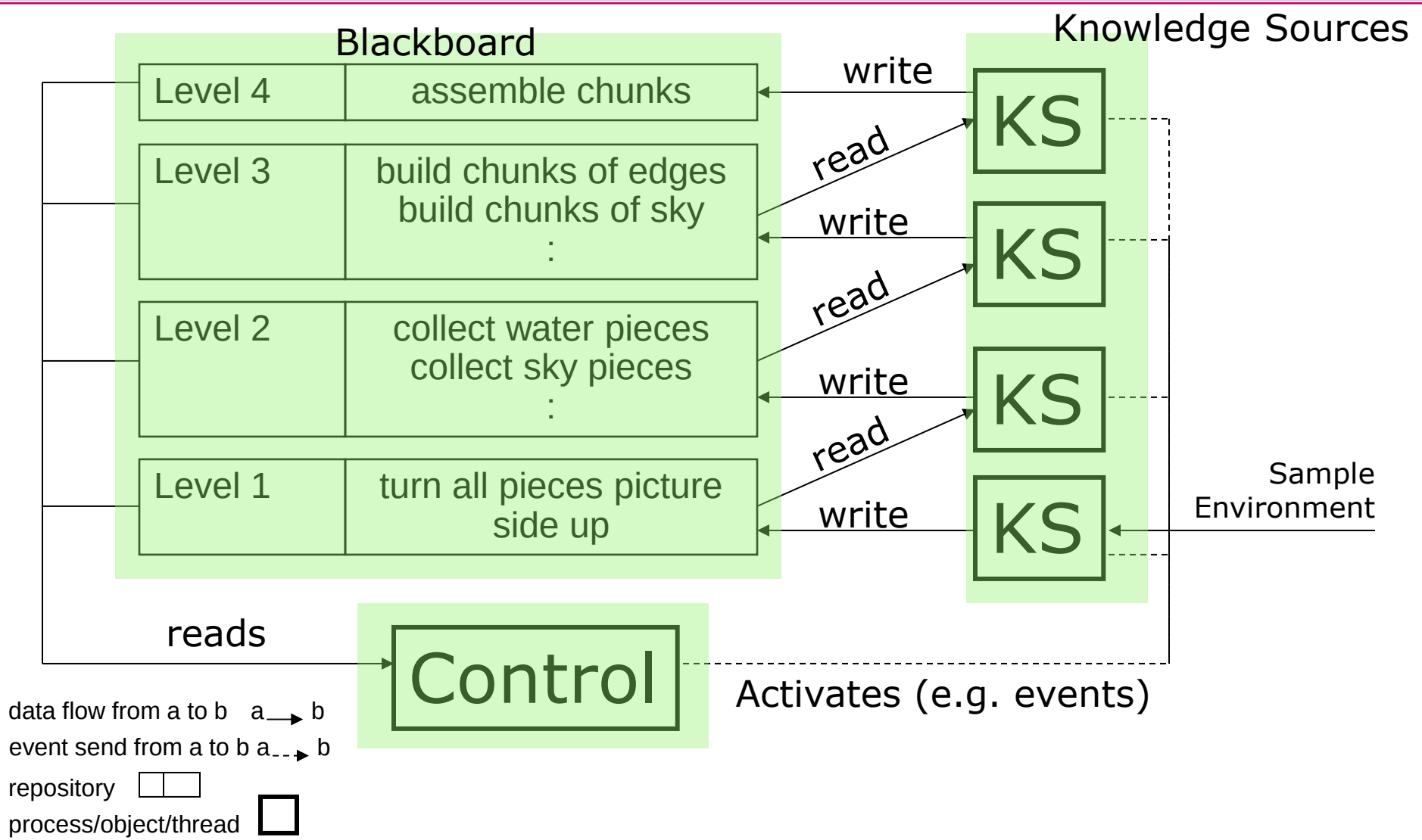
Blackboard 패턴

How do you solve a puzzle?

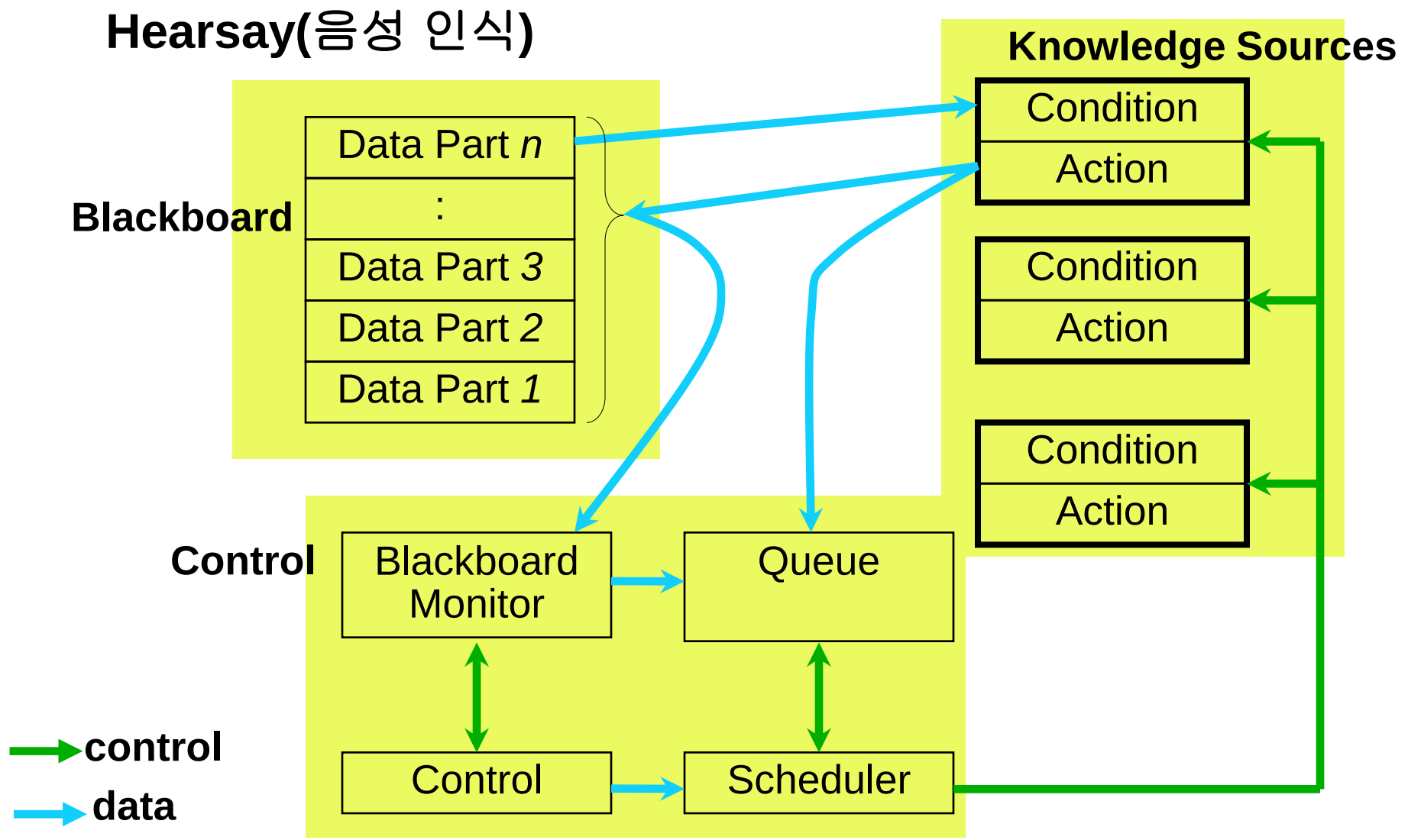
Level 4	assemble chunks
Level 3	build chunks of edges build chunks of sky :
Level 2	collect water pieces collect sky pieces :
Level 1	Turn all pieces picture side up



Blackboard 패턴



Blackboard 패턴



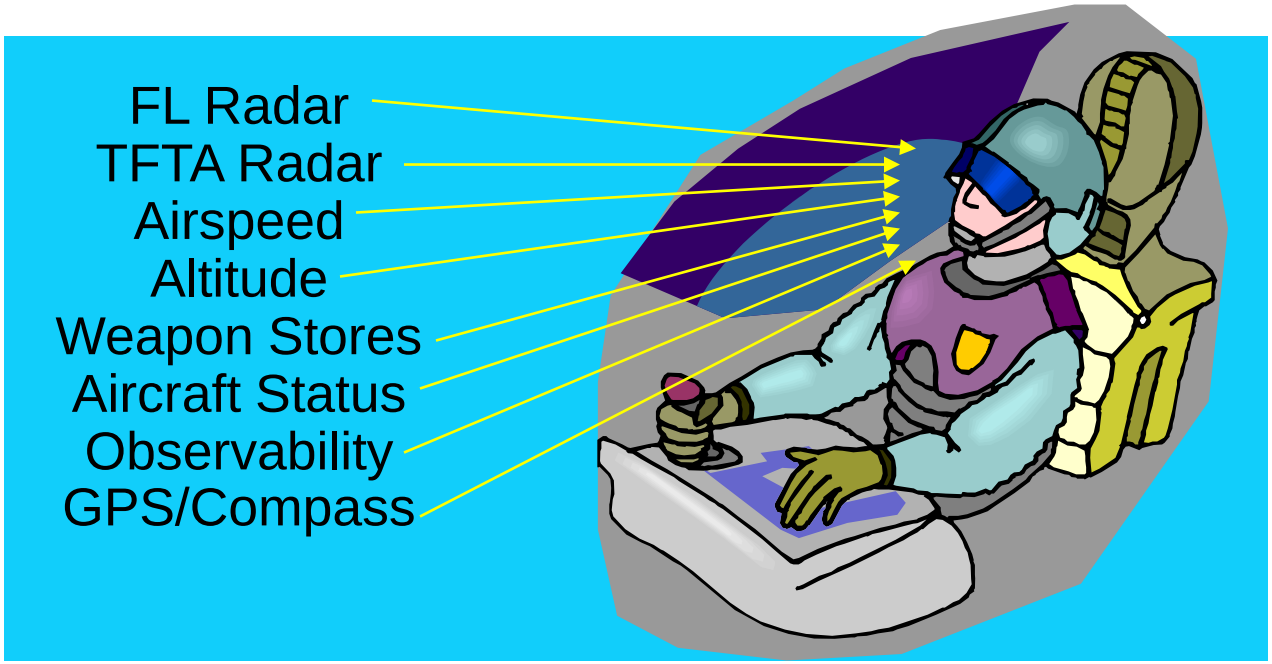
Blackboard 패턴

❖ F-22 Sensor Fusion System

- 비행기 조종사는 넘쳐나는 데이터로 인해 많은 스트레스를 받는다.
  - avoid being detected
  - engage enemy fighters
  - engage ground targets
  - avoid terrain obstacles
  - avoid surface-to-air ordinance
  - navigate and find the targets
  - oh yeah,... and fly the airplane too!

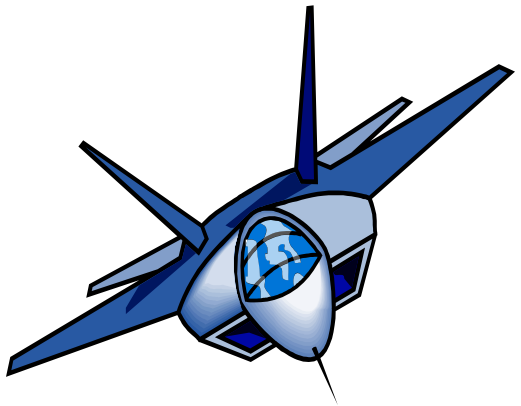
Blackboard 패턴

- 기존 시스템은 센서, 표시판, 조절 기능이 분리되었음
- 조종사가 데이터를 조합하고 판단하여야 함



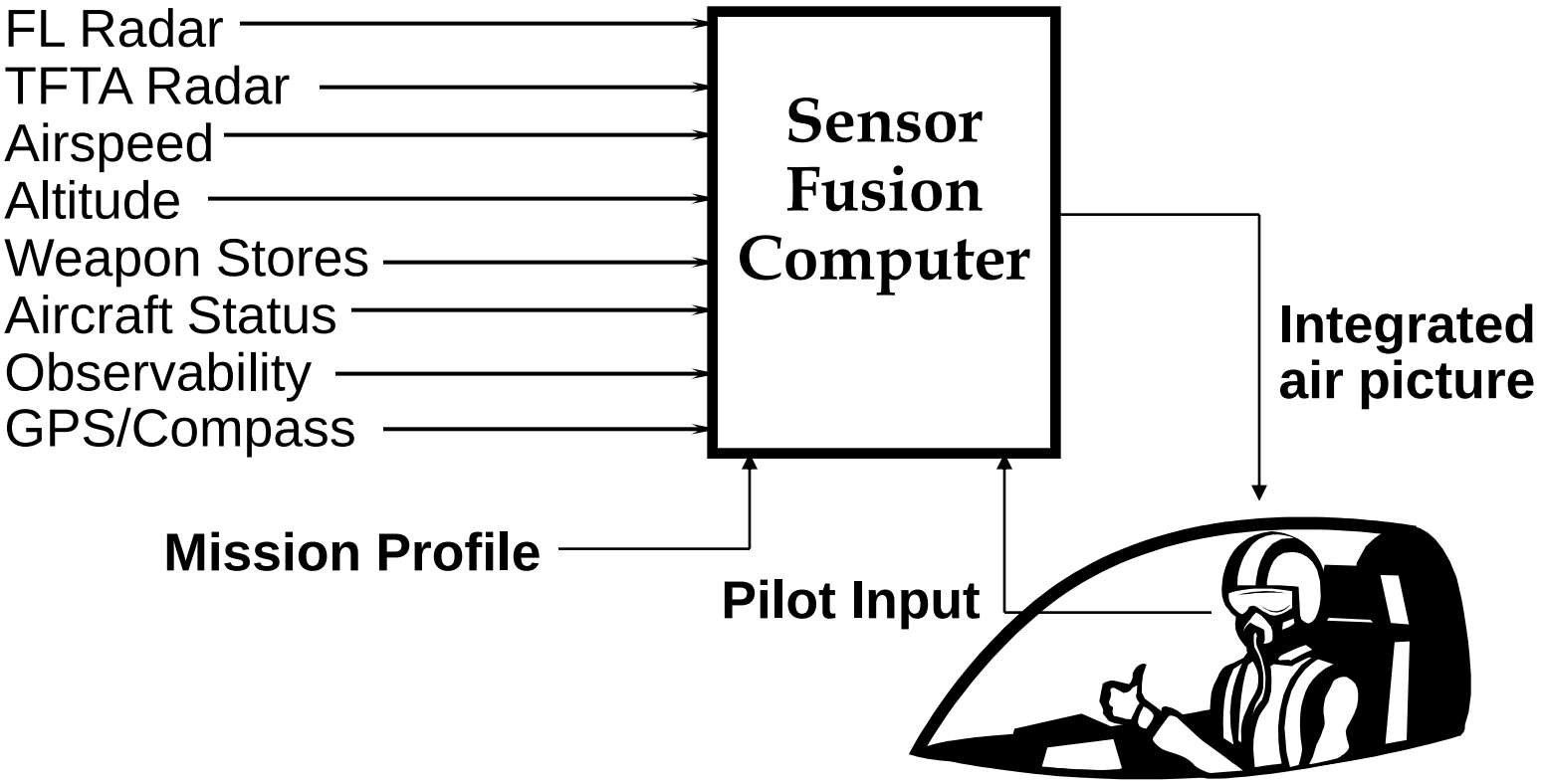
Blackboard 패턴

- F-22 Advanced Tactical Fighter's Sensor Fusion System
  - 목적
    - 데이터 통합
    - 필요 데이터만 선별적으로 보드에 표시
    - 보기 좋은 형태로 데이터를 표시
  - F-22 비행기에서 가장 복잡한 기계 중에 하나로서 blackboard 패턴을 사용

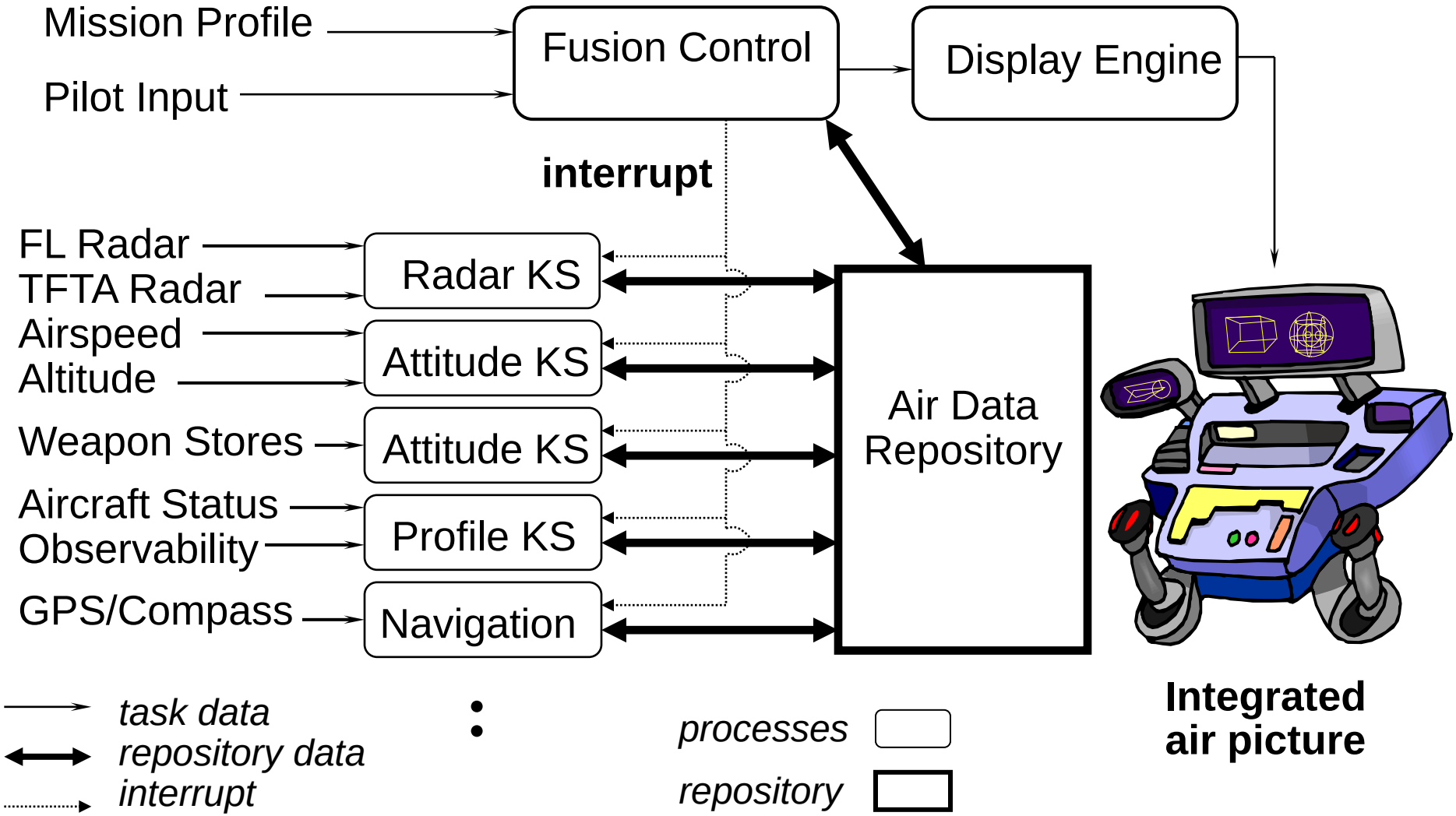


Blackboard 패턴

❖ *"The F-22 Advanced Tactical Fighter's Sensor Fusion System 은 조종사가 필요로 하는 정보를 하나의 계기판에 표시해 준다."*



Blackboard 패턴



Blackboard 패턴

❖ 장점

- 완벽한 해법을 찾기 어려운 경우에 사용할 수 있음
- KS, Control, Blackboard 가 독립적으로 동작하여 가변성이나 유지보수성이 좋음
- KS는 타 문제 도메인에 재사용될 수 있음

❖ 단점

- 완벽한 해법을 제시하지 못하므로 얼마 동안 동작해야 하는지 알 수가 없음(성능 문제)
- 계산 결과가 항상 동일하지 않아 테스트가 어려움
- 많은 시간에 걸쳐 수정되어야 하므로 개발에 많은 노력이 필요

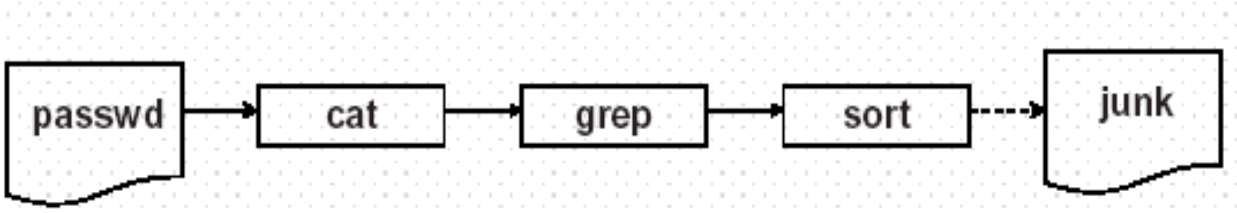
Pipes and Filters 패턴

❖ 정의

- 데이터 스트림을 처리하는 패턴
- 데이터는 Pipe를 통해서 Filter로 전달
- 전달된 데이터는 Filter를 통해 걸러지고 pipe를 통해 다음 Filter로 이동

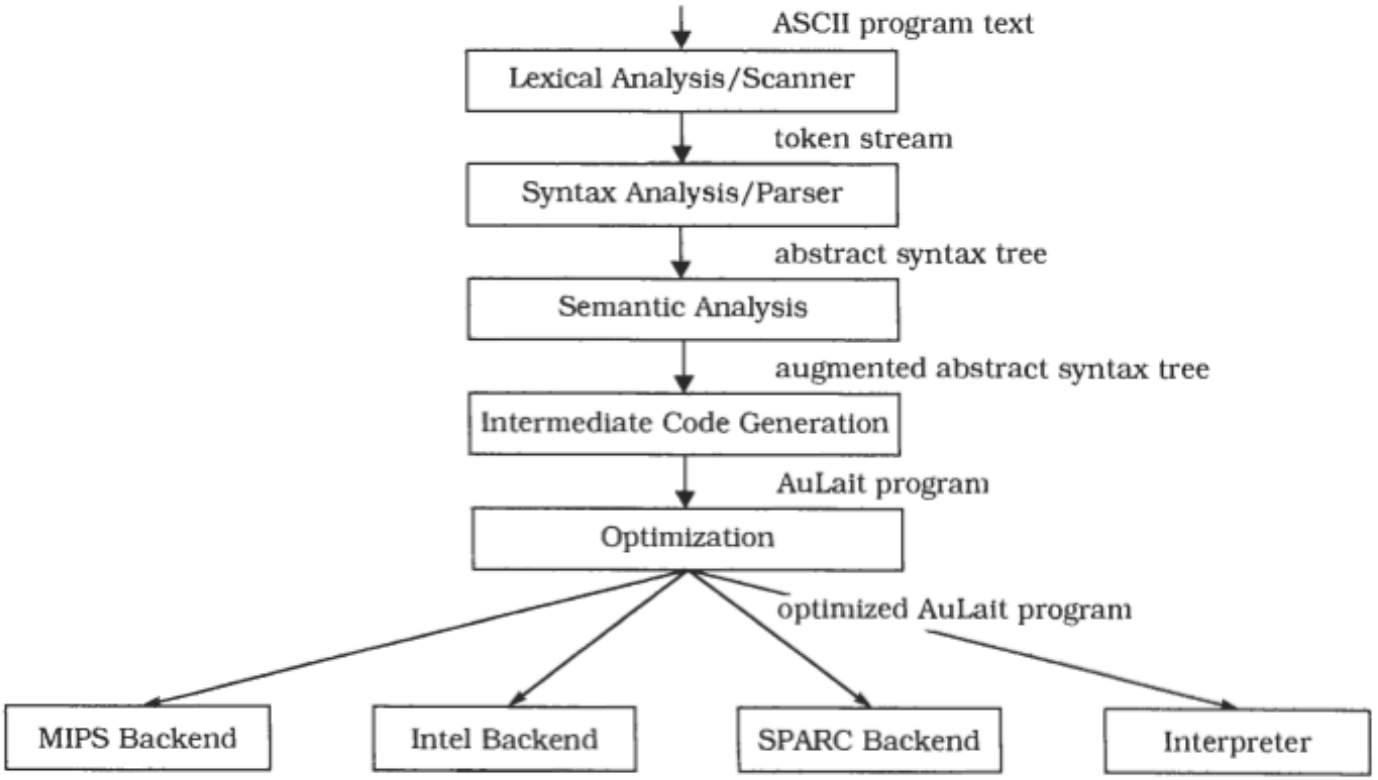
❖ 예제

- Unix command system
  - Ex>> cat etc/passwd | grep “joe” | sort > junk



Pipes and Filters 패턴

❖ Programming Language Compiler



Pipes and Filters 패턴

- 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
- 해법(Solution)
  - 입력된 데이터를 처리하는 단계를 순차적(sequencial)으로 처리
  - 하나의 처리 단계(Filter로 구현)에서 처리된 결과물은 다음 처리단계의 입력물
  - 각각의 처리 단계를 Pipe로 연결

Pipes and Filters 패턴

❖ 구성 컴포넌트

- Filters
  - 데이터가 입력되면 시작
  - 데이터를 보강하거나 정제, 변형하는 프로세스를 진행
- Pipes
  - 데이터가 흐르는 공간
  - 필터와 필터 사이를 연결
    - 데이터 소스와 최초 필터간의 역할
    - 최종 필터와 데이터 싱크 간의 연결
  - 연결된 Filter에게 데이터를 입력하고 출력하는 역할
- Data Source
  - 시스템에 들어오는 입력 값
  - 동일한 구조체나 타입으로 이뤄진 일련의 데이터 값
- Data Sink
  - Pipe를 통해 나오는 최종 결과물을 취합

Pipes and Filters 패턴

❖ 설계 순서

1. 처리 단계 순서에 맞게 시스템의 작업을 나눈다.
   - 각 단계는 이전 단계의 아웃풋에만 의존성을 가져야 한다.
2. 데이터 포맷을 결정한다.
3. 파이프 간의 연결 방법을 설계한다.
4. 필터 설계하고 구현한다.
5. 에러 핸들링을 설계하고 구현한다.
6. 파이프라인을 구현한다.

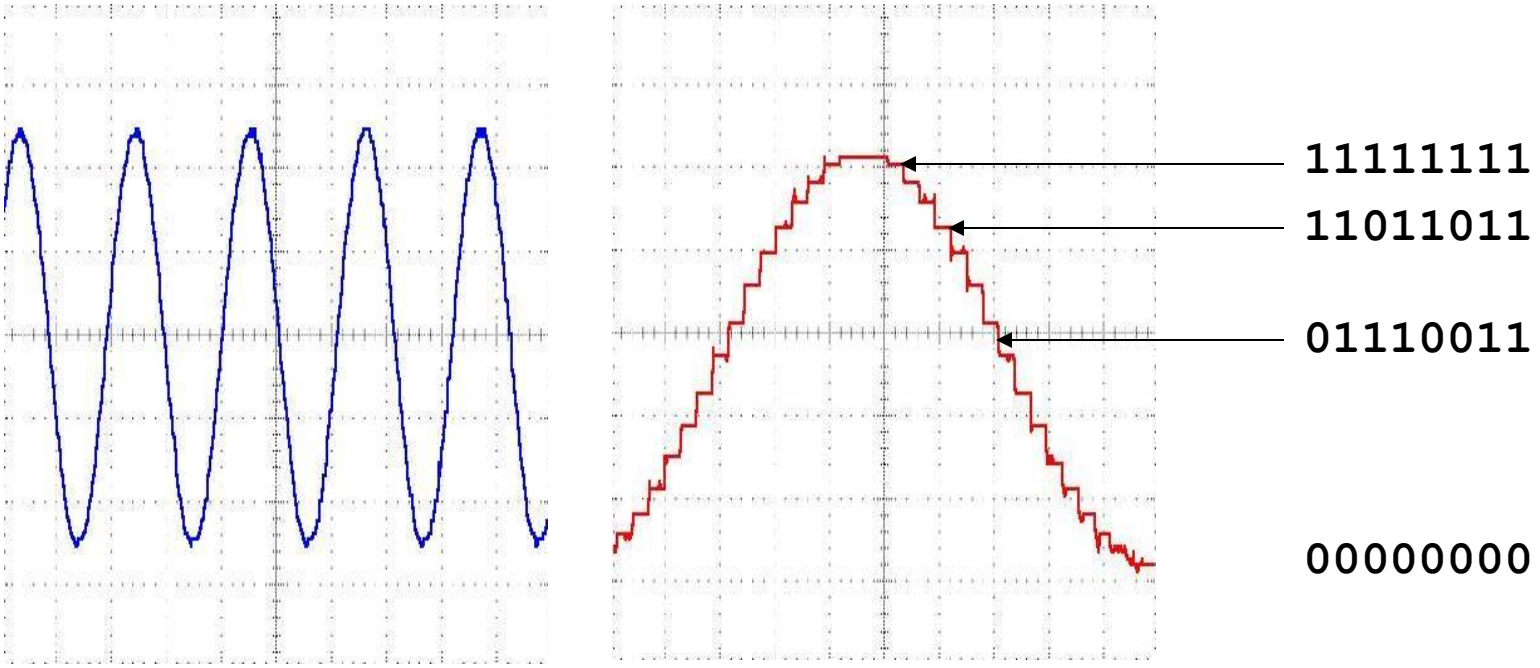
Pipes and Filters 패턴

- Pipes and Filters 패턴은 시그널 처리와 관련된 어플리케이션에 주로 사용된다.
  - radar
  - medical
  - process control
  - sonar
  - audio
  - video
  - telemetry 등등

Pipes and Filters 패턴

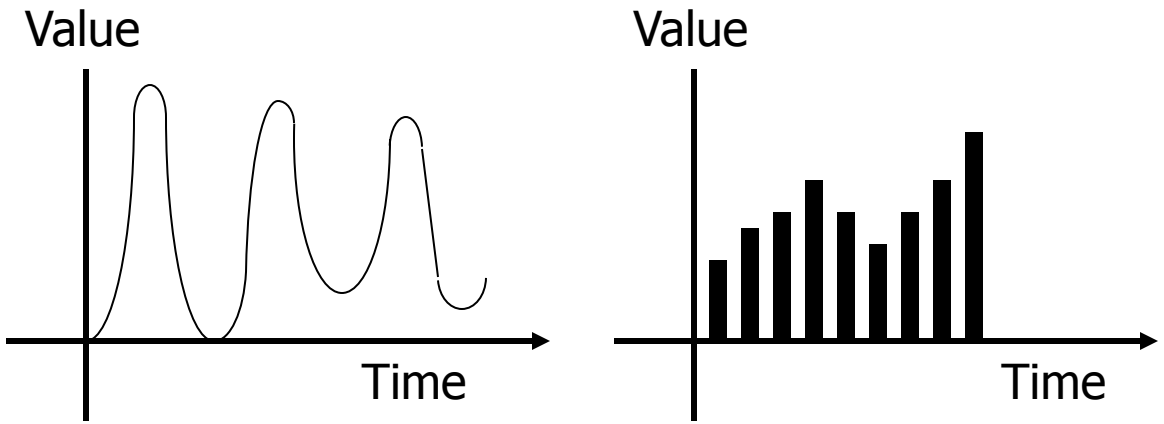
❖ Signaling Process

- 최근에 주로 사용하는 시그널 처리는 지속적으로 들어오는 아날로그 신호를 디지털 신호로 바꿔주고 다시 디지털 신호를 아날로그 신호로 바꿔주는 방식으로 개발된다.



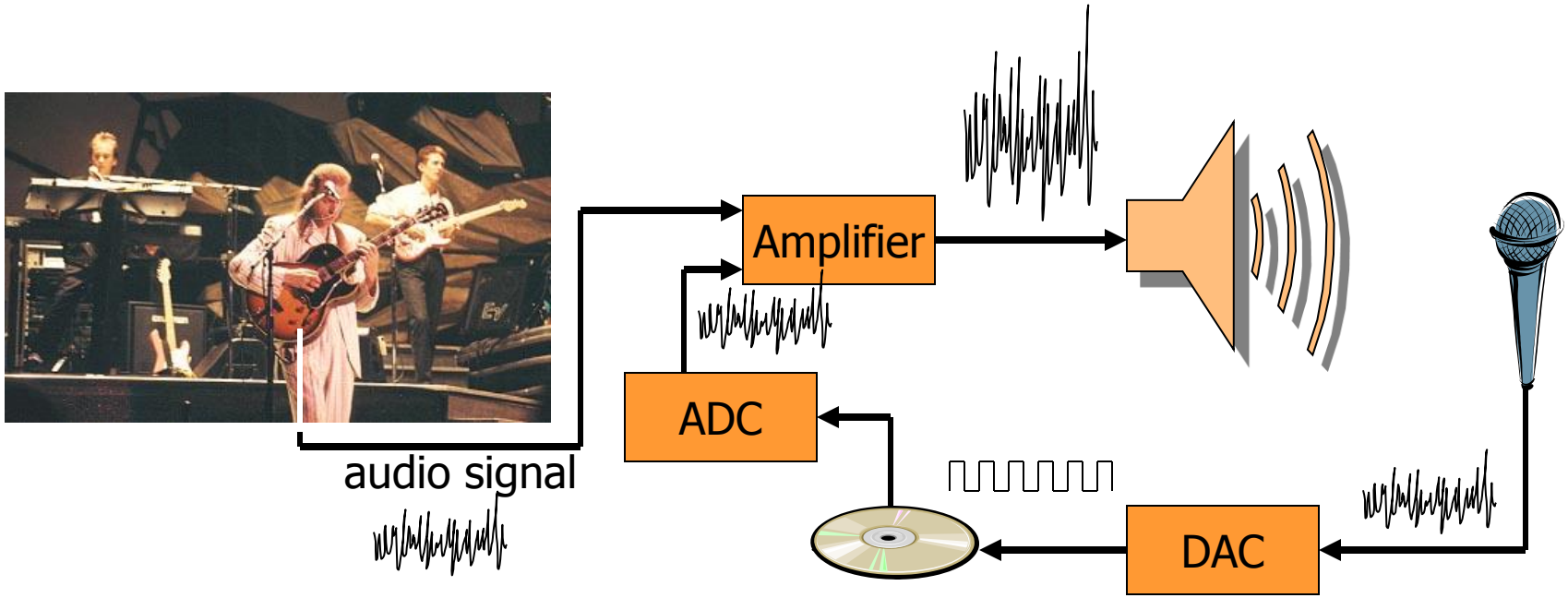
❖ Digital Analog

- An analog quantity is one that has a continuous value over time.
- A digital quantity is on that has a discrete set of values over time.



Pipes and Filters 패턴

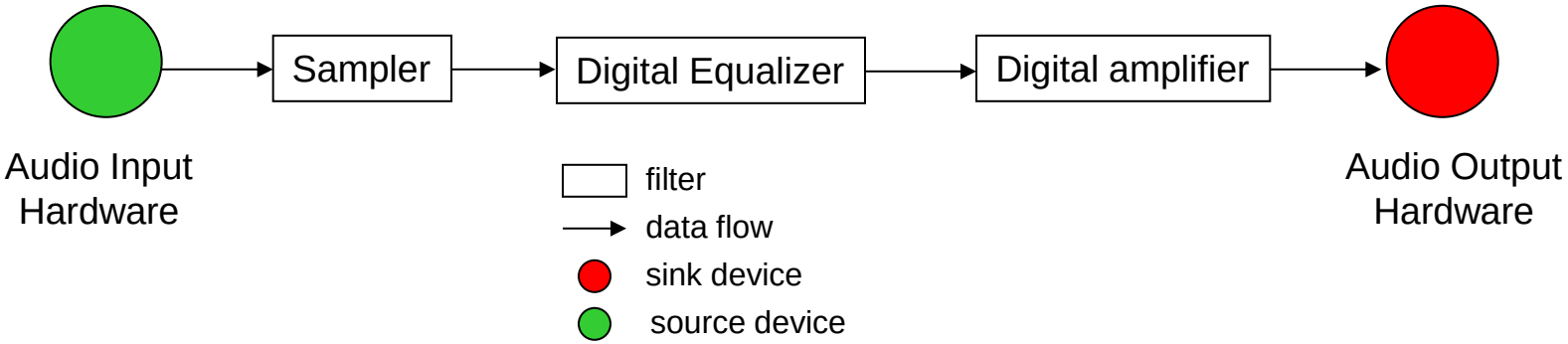
- Digital Audio
  - Analog to Digital Converters (ADC) -> analog voltages to digital values.
  - Digital to Analog Converters (DAC) -> digital values to voltages.



Pipes and Filters 패턴

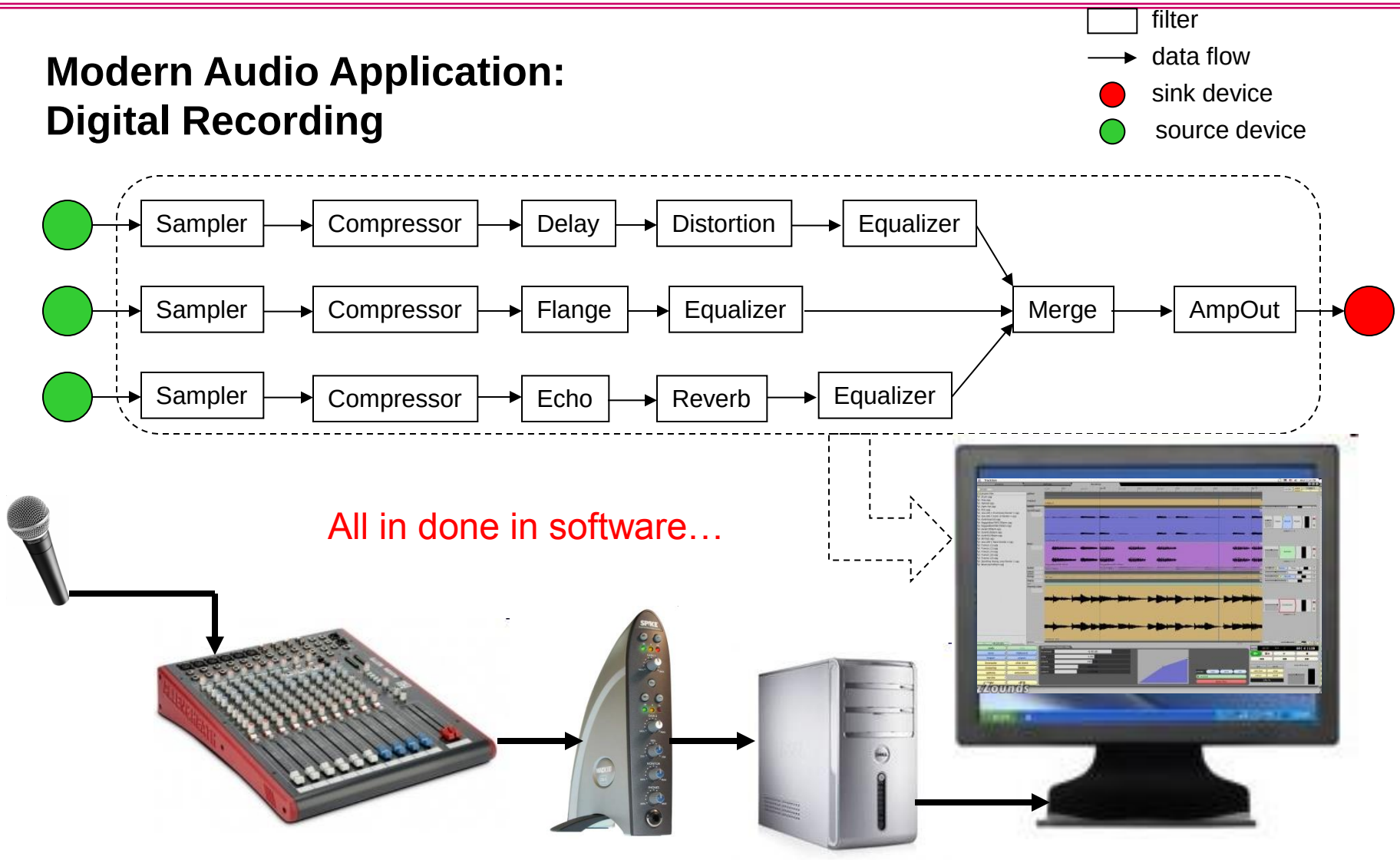
- 1950년대에 Max Mathews는 unit generators (UG)라는 네트워크 모듈을 이용하여 신디사이저를 개발했다.

- UG는 신디사이저의 핵심으로 다음과 같은 기능을 한다.
  - 사운드 재생과 사운드처리에 사용(최근까지 사용)
  - 처음에는 하드웨어로 개발되었으나 최근에는 소프트웨어로 개발
- 이 기술은 최근에 음악 기기, 레코더, 음악 재생 기계(CDs, MP3 players) 등에 사용되고 있다.



Pipes and Filters 패턴

Modern Audio Application: Digital Recording



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

Pipes and Filter 패턴

❖ 장점
- 중간 결과 파일이 불 필요함
- Filter의 교환이나 재조합이 쉬움
- Filter의 재사용성이 좋음
- Parallel 프로세싱에 용이함

❖ 단점
- 다음과 같은 처리 시간이 성능에 악영향
  - Filter 간의 전송 시간
  - Context switching 시간
  - Synchronization
- 에러 처리가 어려움 (데이터 복구 작업 등)

Broker 패턴

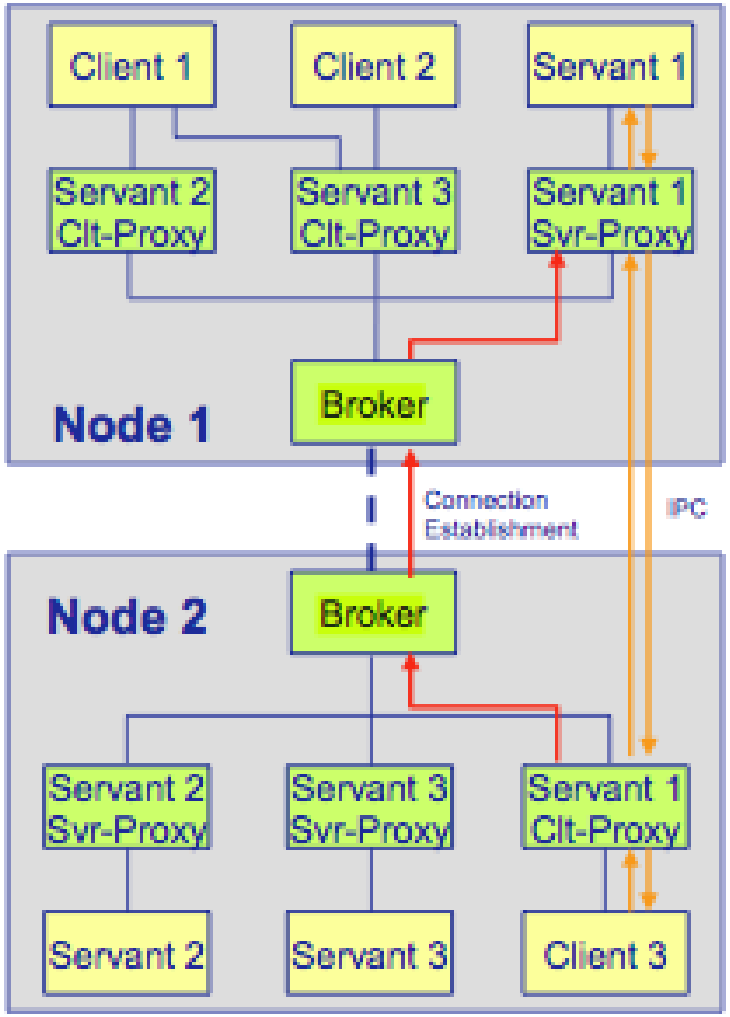
❖ 정의

- 외부에 분산된 컴포넌트를 호출하려고 할 때 클라이언트 요청 값을 분석하여 서버 컴포넌트에 전달하고 그 결과값을 전달하는 역할을 하는 패턴
- 클라이언트와 서버 사이의 브로커라는 컴포넌트를 두어 보다 효과적으로 서버와 클라이언트 사이를 분리할 수 있어 분산 시스템을 구축하는데 용이함

❖ 예제

- 광역 네트워크 기반의 CIS(city information system) 시스템
- CORBA (Common Object Request Broker Architecture)

Broker 패턴


Client 1
Client 2
Servant 1
Servant 2
Clt-Proxy
Servant 3
Clt-Proxy
Servant 1
Svr-Proxy
Broker
Node 1
Connection
Establishment
IPC
Node 2
Broker
Servant 2
Svr-Proxy
Servant 3
Svr-Proxy
Servant 1
Clt-Proxy
Servant 2
Servant 3
Client 3

Broker 패턴

❖ 정황(Context)

- 독립적인 컴포넌트 형태로 이질적인 환경에서 작동하는 분산 시스템을 개발하는 경우

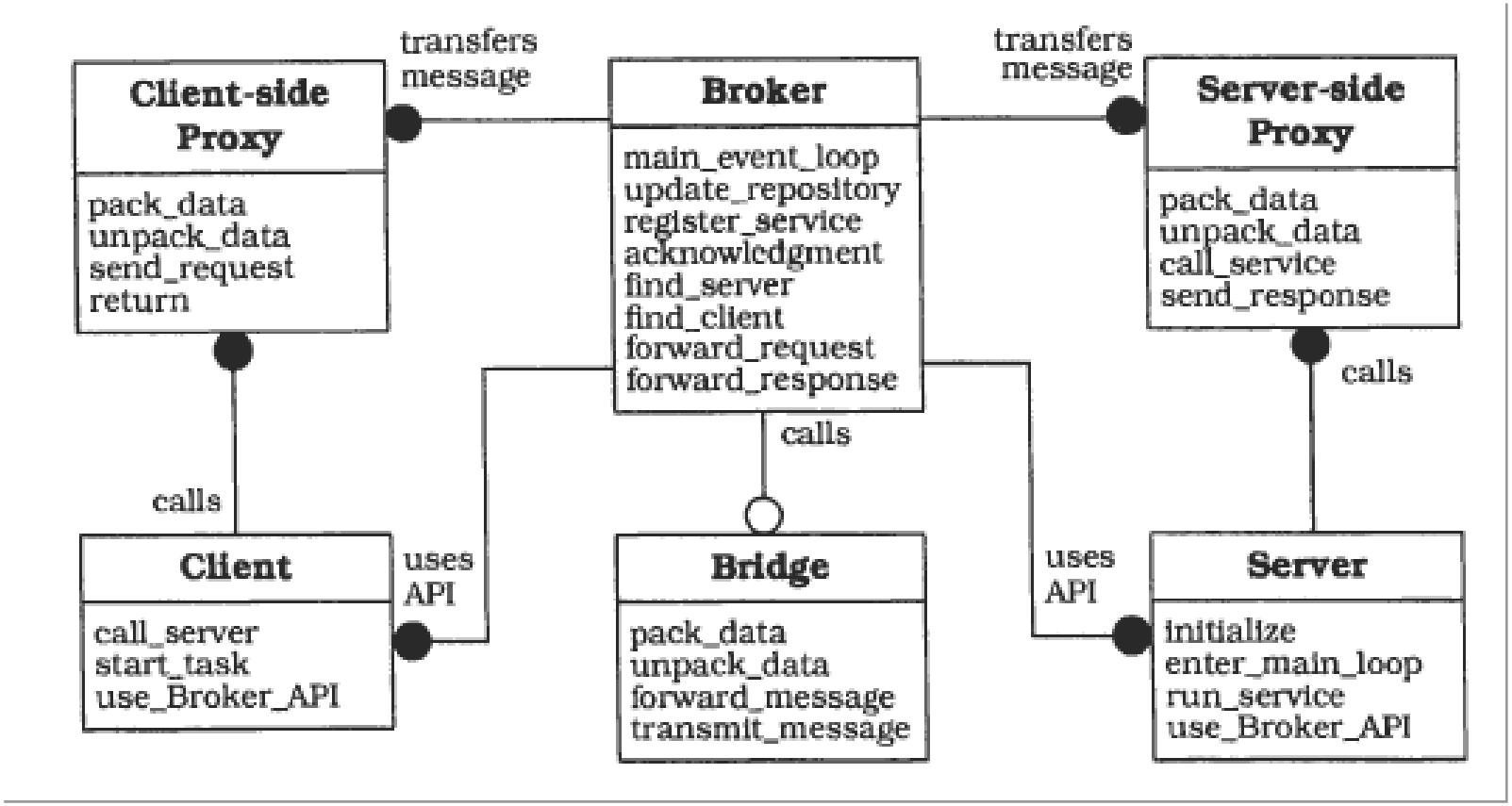
❖ 문제(Problem)

- 독립 컴포넌트마다 실행 환경이 다르고 이들끼리 통신이 필요한 경우
- 클라이언트와 서버들이 추가, 삭제 및 변경이 자주 일어날 경우

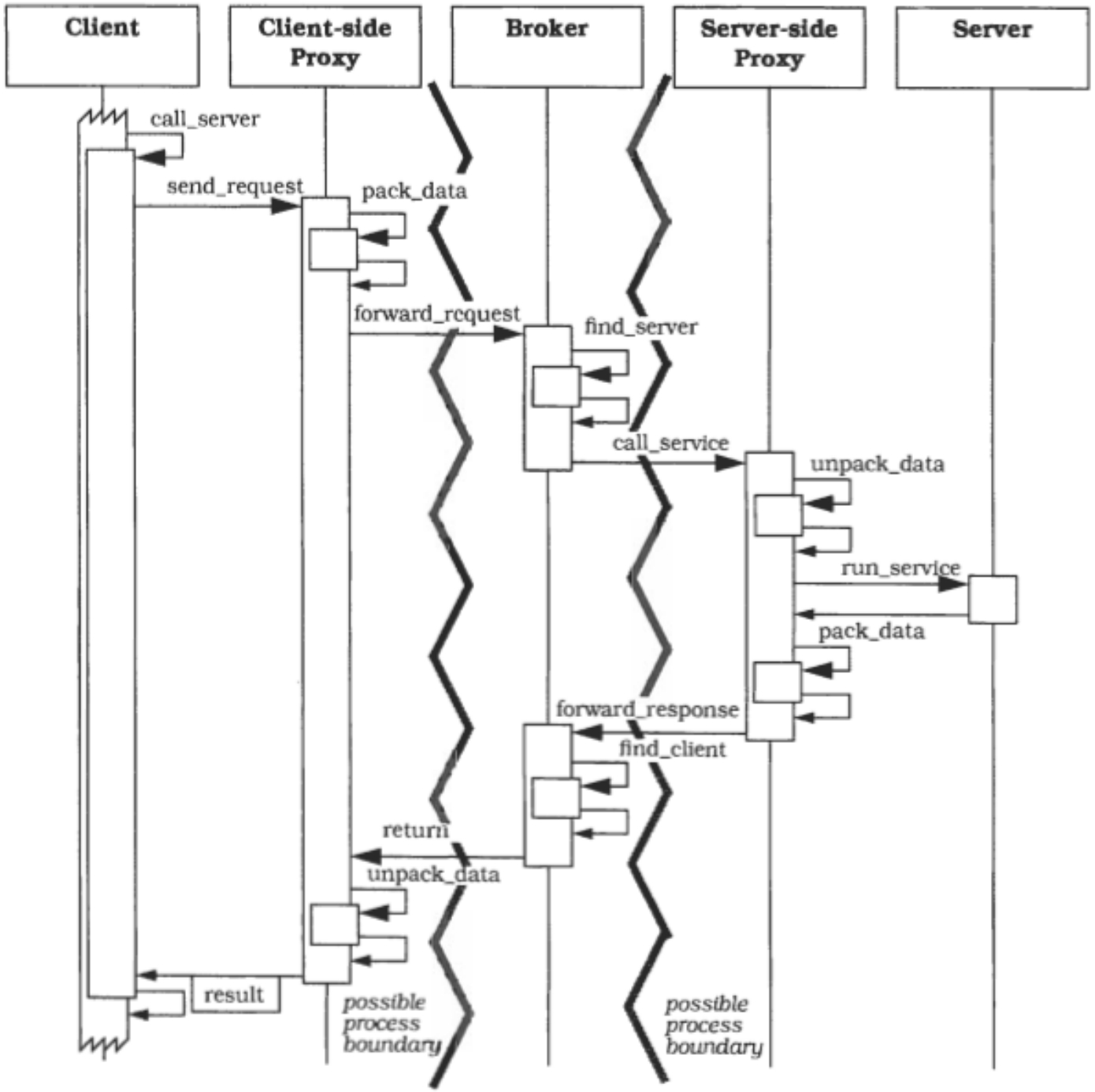
❖ 해법(Solution)

- Broker 컴포넌트를 도입하여 클라이언트와 서버 사이를 분리
- Broker 컴포넌트가 클라이언트와 서버의 정보를 가지고 있어 서로간의 통신을 조율
- 클라이언트와 서버 Proxy를 두어 특정 환경과 관련된 부분을 처리
- Bridge를 두어 네트워크 통신과 관련된 부분을 이관하여 처리

Broker 패턴


Client-side Proxy
pack_data
unpack_data
send_request
return
transfers message
Broker
main_event_loop
update_repository
register_service
acknowledgment
find_server
find_client
forward_request
forward_response
transfers message
Server-side Proxy
pack_data
unpack_data
call_service
send_response
calls
calls
calls
Client
call_server
start_task
use_Broker_API
uses API
Bridge
pack_data
unpack_data
forward_message
transmit_message
uses API
Server
initialize
enter_main_loop
run_service
use_Broker_API

Broke

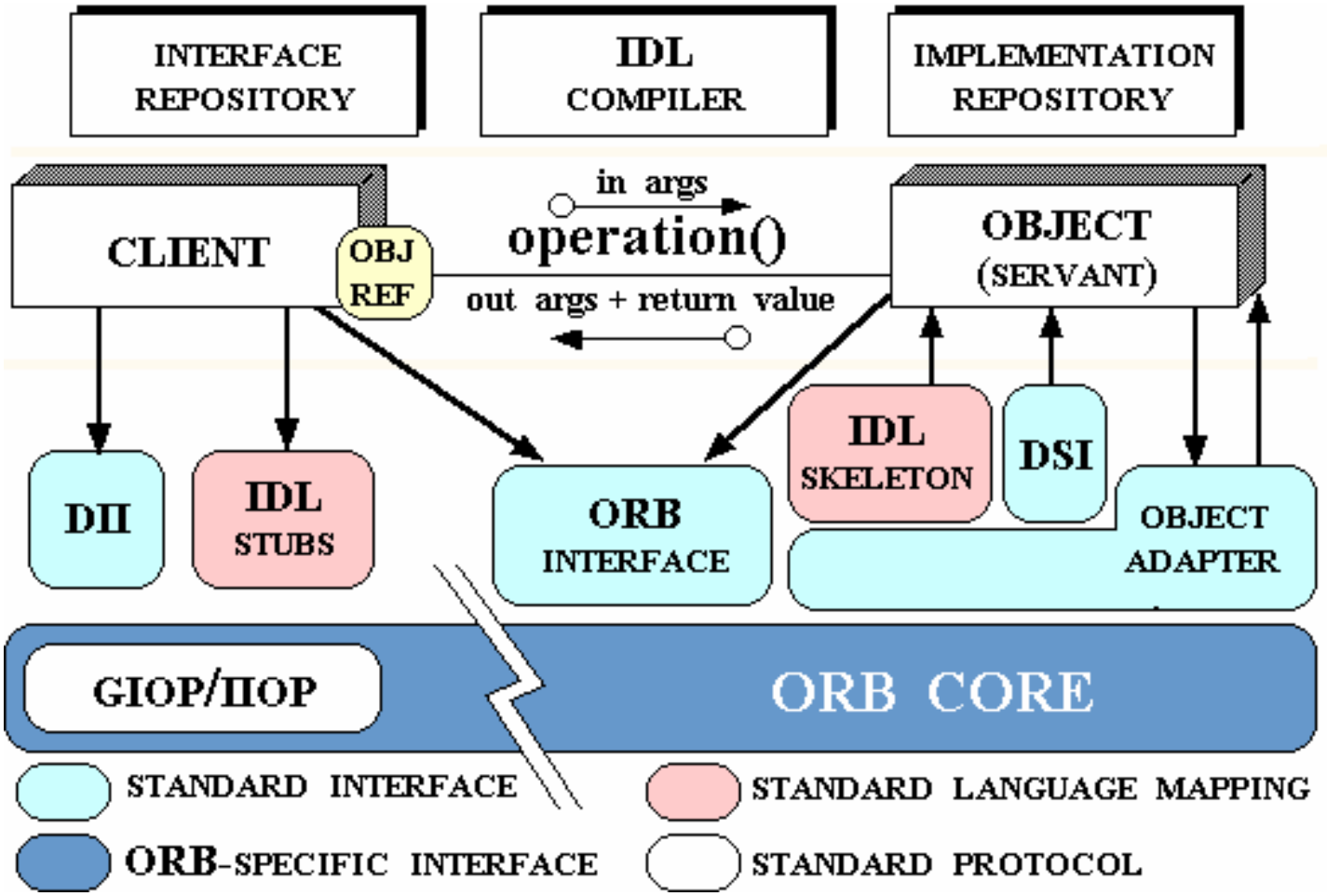

Client
Client-side Proxy
Broker
Server-side Proxy
Server
call_server
send_request
pack_data
forward_request
find_server
call_service
unpack_data
run_service
pack_data
forward_response
find_client
return
unpack_data
result
possible process boundary
possible process boundary

Broker 패턴

❖ 설계 순서

1. 객체 모델을 정의하거나 기존 모델을 재사용 할 지 결정
2. 컴포넌트들 사이의 상호 연동을 어떤 방식으로 할 지 결정
3. 클라이언트와 서버 간의 협력을 위한 Broker 컴포넌트의 역할 정의
4. Proxy 객체를 사용해 환경과 관련된 부분 캡슐화 설계
5. Broker 컴포넌트 설계
6. IDL 컴파일러 설계

Broker 패턴


INTERFACE REPOSITORY
IDL COMPILER
IMPLEMENTATION REPOSITORY
CLIENT
OBJ REF
in args
operation()
out args + return value
OBJECT (SERVANT)
DII
IDL STUBS
ORB INTERFACE
IDL SKELETON
DSI
OBJECT ADAPTER
GIOP/IIOP
ORB CORE
STANDARD INTERFACE
STANDARD LANGUAGE MAPPING
ORB-SPECIFIC INTERFACE
STANDARD PROTOCOL

Broker 패턴

❖ Server Code

```
...
// Create an object request broker
ORB orb = ORB.init(args, null);

// Create a new address book ...
AddressBookServant servant = new AddressBookServant();

// ... and connect it to our orb
orb.connect(servant);

// Obtain reference for our nameservice
org.omg.CORBA.Object object = orb.resolve_initial_references("NameService");

// Since we have only an object reference, we must cast it to a NamingContext. We use a helper class for this purpose
NamingContext namingContext = NamingContextHelper.narrow(object);

// Add a new naming component for our interface
NameComponent list[] = { new NameComponent("address_book", "") };

// Now notify naming service of our new interface
namingContext.rebind(list, servant);
```

Broker 패턴

❖ Client Code

```
// import servant class that is generated by IDL  converter
import  address_book_system.address_bookPackage.*;

...
// Create an object request broker
 ORB orb = ORB.init(args, null);

 // Obtain object reference for name service ...
  org.omg.CORBA.Object object =
  orb.resolve_initial_references("NameService");

  // ... and narrow it to a NameContext
  NamingContext namingContext =
  NamingContextHelper.narrow(object);

  // Create a name component array
  NameComponent nc_array[] =
  { new NameComponent("address_book","") };

  // Get an address book object reference ...
  org.omg.CORBA.Object objectReference =
  namingContext.resolve(nc_array);

  // ... and narrow it to get an address book
  address_book AddressBook =
  address_bookHelper.narrow(objectReference);

  // call the  address book interface
  name = AddressBook.name_from_email(email);
```

Broker 패턴

❖ EJB3.0 Code

❖ Server

```
...
@Stateless(name="HelloBean")
@Remote(HelloRemote.class)

public class HelloWorldBean {
    public String sayHello() {
        return "Hello World!!!";
    }
}
```

❖ Client

```
...

InitialContext ic = new InitialContext();
Hello = (HelloRemote) ic.lookup("example/HelloBean/remote");

...
```

Broker 패턴

❖ 장점
- 컴포넌트간의 위치 투명성을 제공
- 플랫폼 간의 Portability 제공함
- 서버 다른 시스템의 연동을 용이하게 함
- 재사용 컴포넌트 확보에 용이

❖ 단점
- 성능에 대한 불 이익
- 장애 대처율이 떨어짐
- 테스트 디버깅의 복잡함 (서버, 클라이언트 연동 시에)
- 분산 환경을 지원하는 시스템이 많지 않음

MVC 패턴

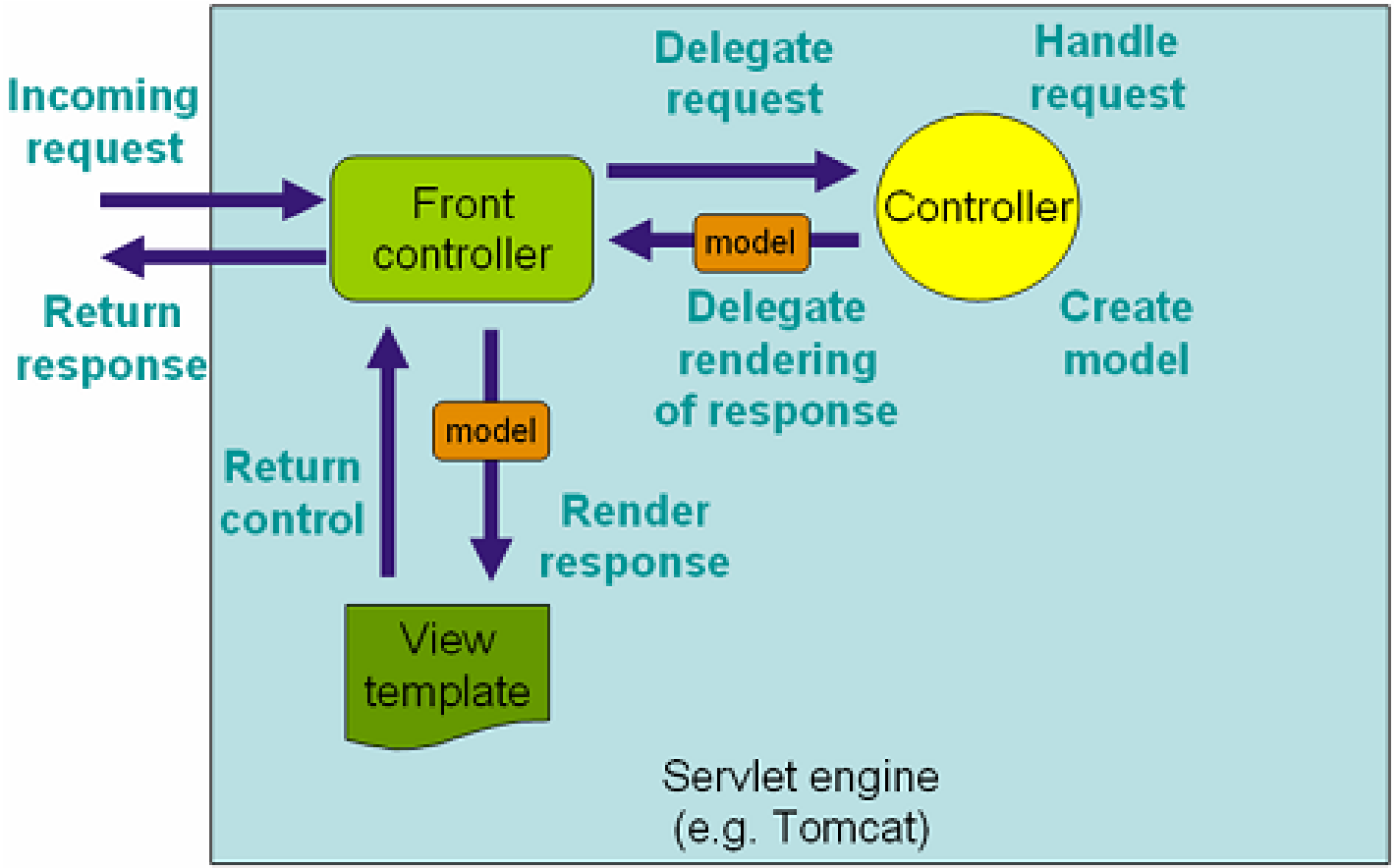
❖ 정의

- 하나의 데이터 값(도메인 오브젝트)을 여러 개의 클라이언트 화면으로 일관적으로 보여줄 수 있는 패턴
- 화면(View)과 데이터 값(Model)의 연결 부분을 컨트롤러(Control)가 관리하여 View의 추가, 변경, 삭제가 Model에 영향을 미치지 않고 Model의 변화도 View에 영향을 미치지 않게 하는 패턴

❖ 예제

- 웹 기반 서비스 시스템 (거의 대부분)
- IOS application 서비스

MVC 패턴


Incoming request
Return response
Front controller
Delegate request
Handle request
Controller
model
Delegate rendering of response
Create model
model
Return control
Render response
View template
Servlet engine (e.g. Tomcat)

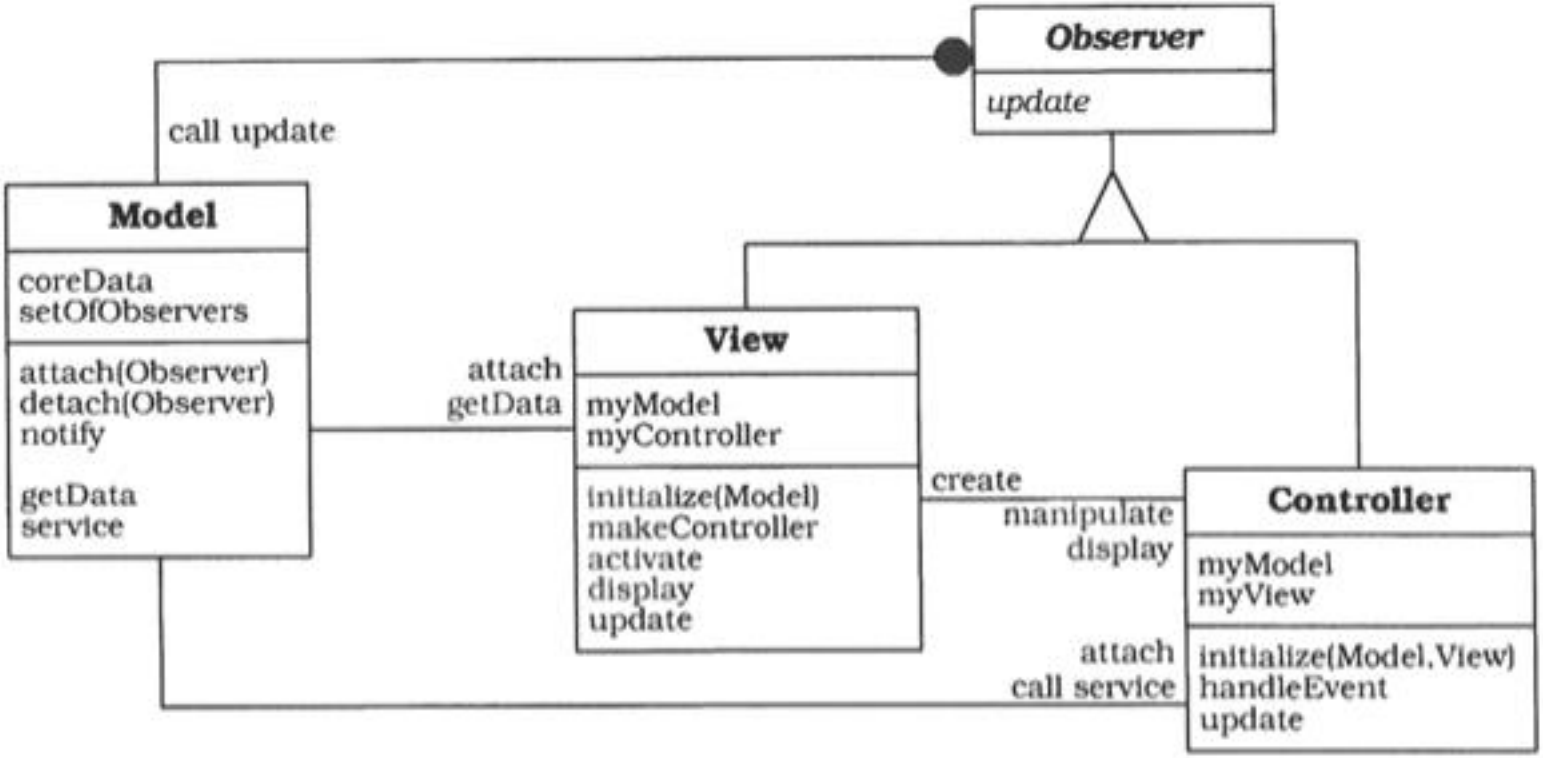
MVC 패턴

- 정황(Context)
  - 상호작용이 많은 시스템에 유연한 HCI(Human-Computer Interface)를 개발하는 경우

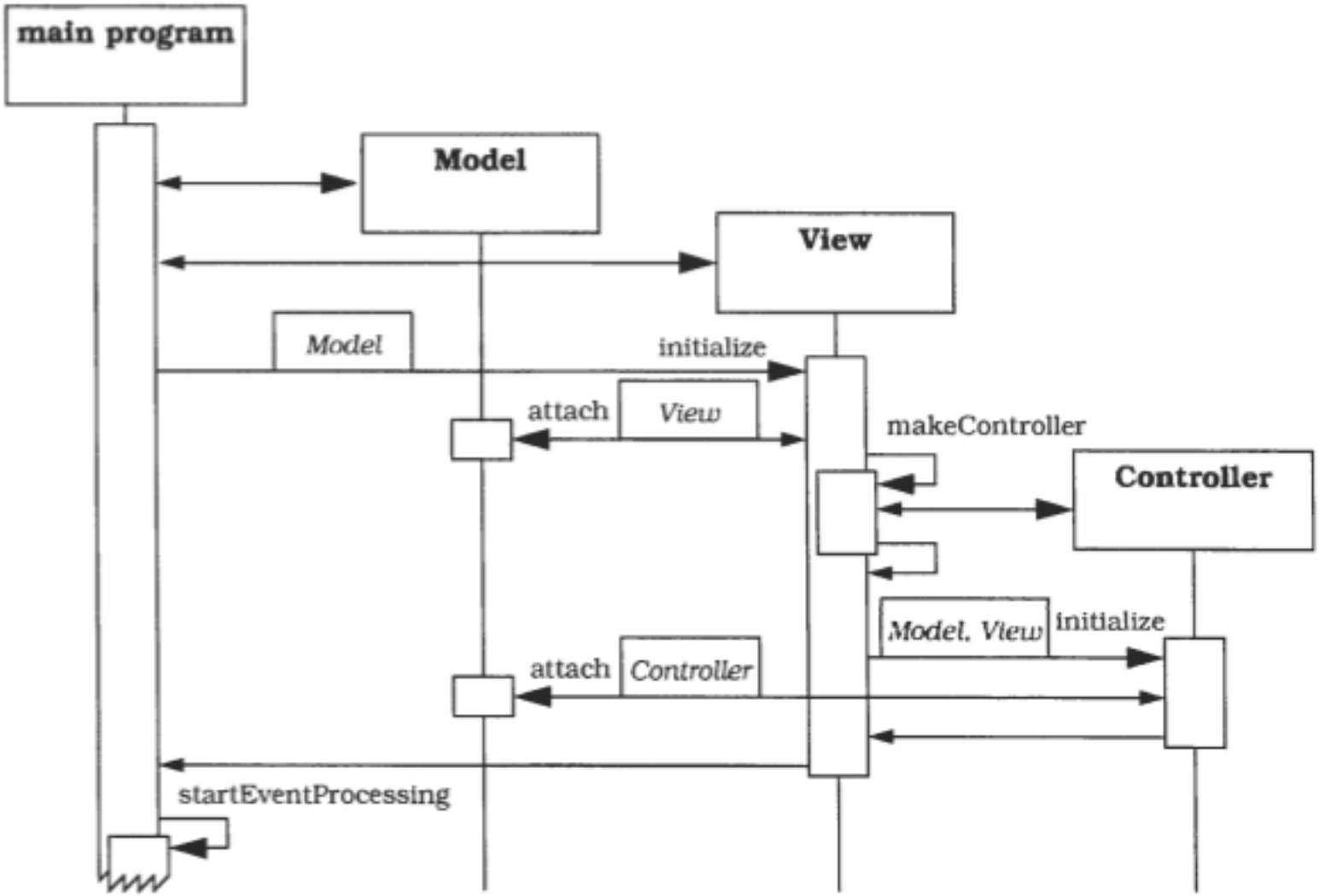
- 문제(Problem)
  - 사용자 인터페이스의 변경이 많이 일어나는 경우
  - 같은 기능을 사용하는 사용자 화면이 다른 경우(예. 마우스로 값 입력 vs. 키보드로 입력)
  - 사용자 인터페이스들 끼리 서로 연관성이 복잡할 경우

- 해법(Solution)
  - Model, View, Controller 이렇게 3개의 영역으로 구분
    - Model : 핵심 데이터와 기능을 캡슐화
    - View : 사용자의 정보를 화면에 표시
    - Controller : View로부터 입력 정보를 받아 관련된 View나 Model을 호출함

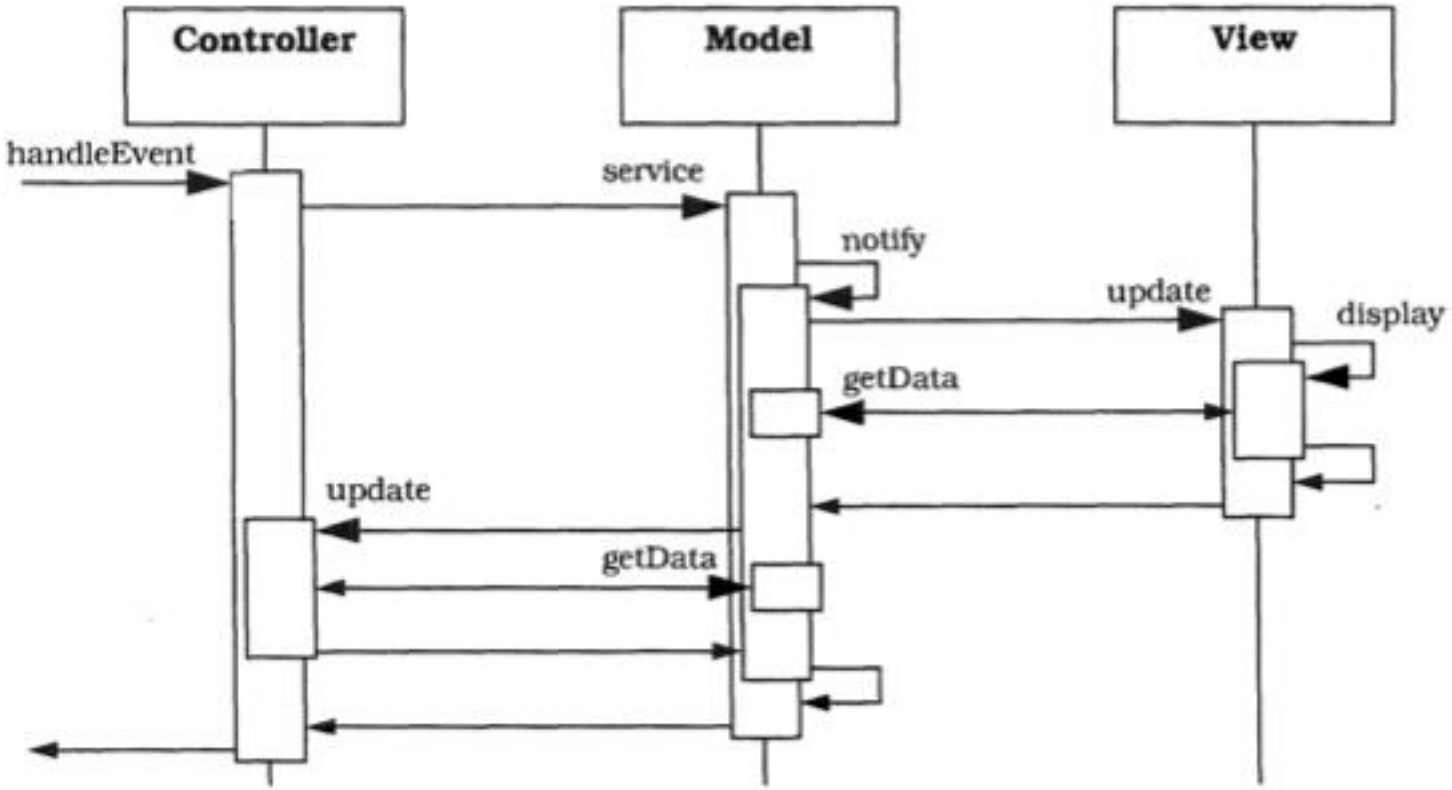
MVC 패턴


Observer
update
call update
Model
coreData
setOfObservers
attach(Observer)
detach(Observer)
notify
getData
service
attach
getData
View
myModel
myController
initialize(Model)
makeController
activate
display
update
create
manipulate
display
Controller
myModel
myView
attach
call service
initialize(Model,View)
handleEvent
update

MVC 패턴


main program
Model
View
Model
initialize
attach
View
makeController
Controller
Model. View
initialize
attach
Controller
startEventProcessing

MVC 패턴


Controller
Model
View
handleEvent
service
notify
update
display
getData
update
getData

MVC 패턴

❖ 구현 순서

1. 서비스의 핵심 기능과 사용자 인터페이스 부분을 분리한다.
2. Publisher-Subscriber (Observer) 패턴을 적용하여 모델을 구현한다.
3. View를 설계하고 구현한다.
4. Controller를 설계하고 구현한다.
5. View와 Controller 관계를 설계하고 구현한다.

MVC 패턴

❖ 장점

- 동일한 모델로부터 여러 View 들을 표현할 수 있음
- View들을 동기화할 수 있음
- View의 추가, 변경, 삭제가 자유로움
- 프레임워크로 확장하여 구현이 가능함

❖ 단점

- 설계가 복잡하고 개발이 어려움
- View와 Controller는 밀접히 관련되어 있음

Publisher-Subscriber 패턴

❖ 정의

- 하나의 Publisher가 다수의 Subscriber에게 상태가 변경되었음을 단방향 전파로 통지하는 패턴
- 협력 컴포넌트들의 상태를 동기화하는데 유용함
- Observer 패턴, Dependents 패턴, Event 패턴으로 사용됨

❖ 예제

- GUI 애플리케이션
  - 사용자의 요청에 따른 화면의 변화(줌인, 포커스, 클릭 등)
- MVC 패턴을 애플리케이션

Publisher-Subscriber 패턴

❖ 정황(Context)

- 한번의 호출로 다수의 협력 컴포넌트의 상태를 변경해야 하는 경우

❖ 문제(Problem)

- 특정 컴포넌트에서 발생하는 상태 변경 정보를 하나 이상의 컴포넌트에서 수신해야 함
- Publisher와 Subscriber는 서로 tightly coupled 되어서는 안된다.

❖ 해법(Solution)

- 하나의 컴포넌트를 Publisher로 두고 상태 변경을 받을 컴포넌트들을 subscriber로 둔다.
- Subscriber는 Publisher에서 제공하는 인터페이스를 통해서 등록한다.
- Publisher의 상태가 변경되면 등록된 Subscriber에게 변경 상태를 전송한다.

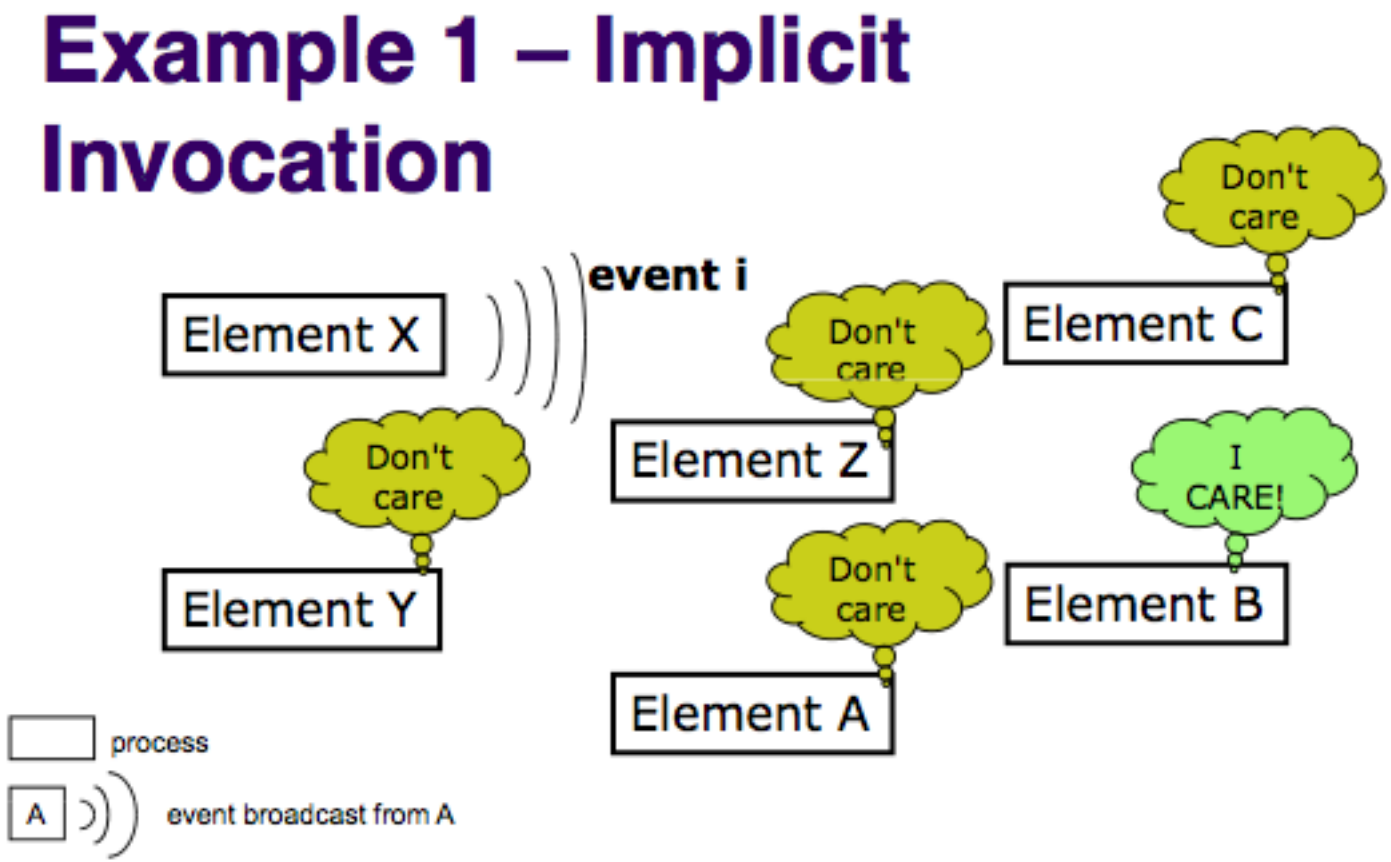
Publisher-Subscriber 패턴

❖ 구현

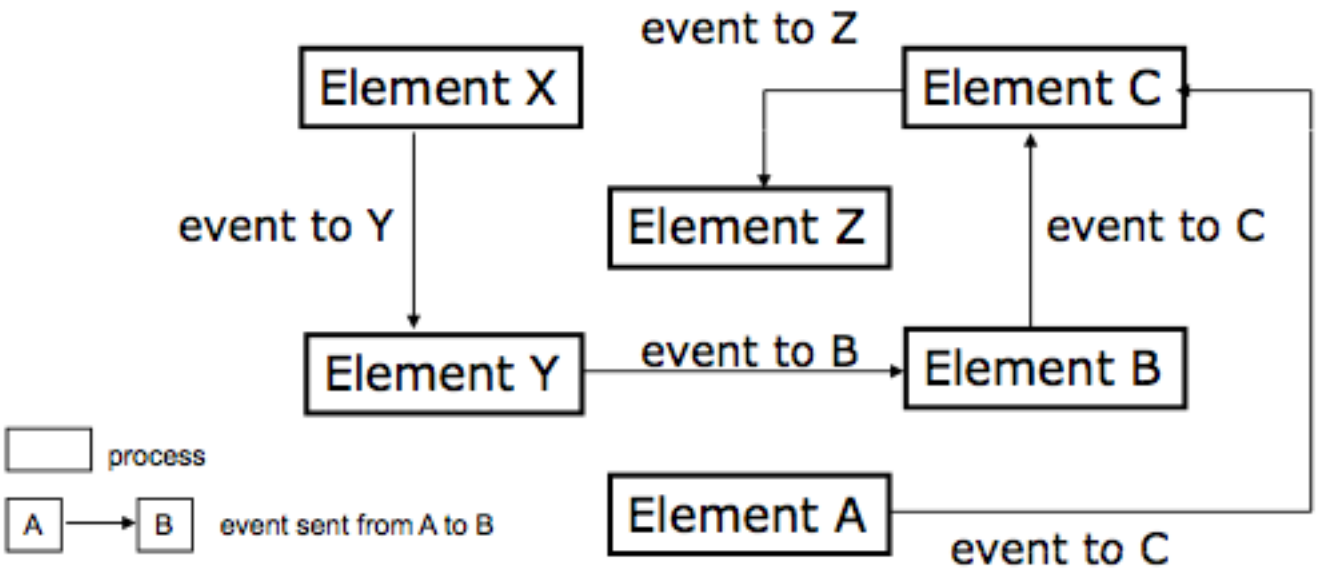
- 이벤트 기반으로 Publisher-Subscriber 패턴을 구현한다.
- 이벤트 기반으로 구현하면 시스템 변경을 쉽게 할 수 있다.
- 이벤트가 명확히 전송되었는지 알 수가 없다. (Non-Deterministic)
- 응답 시간을 명확히 예측할 수 없다. (Non-Deterministic)

❖ 이벤트 기반 시스템

- Explicit Invocation
  - 변경 정보를 전달한 대상을 명확히 알고 변경 정보를 전달함
- Implicit Invocation
  - 변경 정보를 전달할 대상을 명확히 알지 않고 정보를 전달함



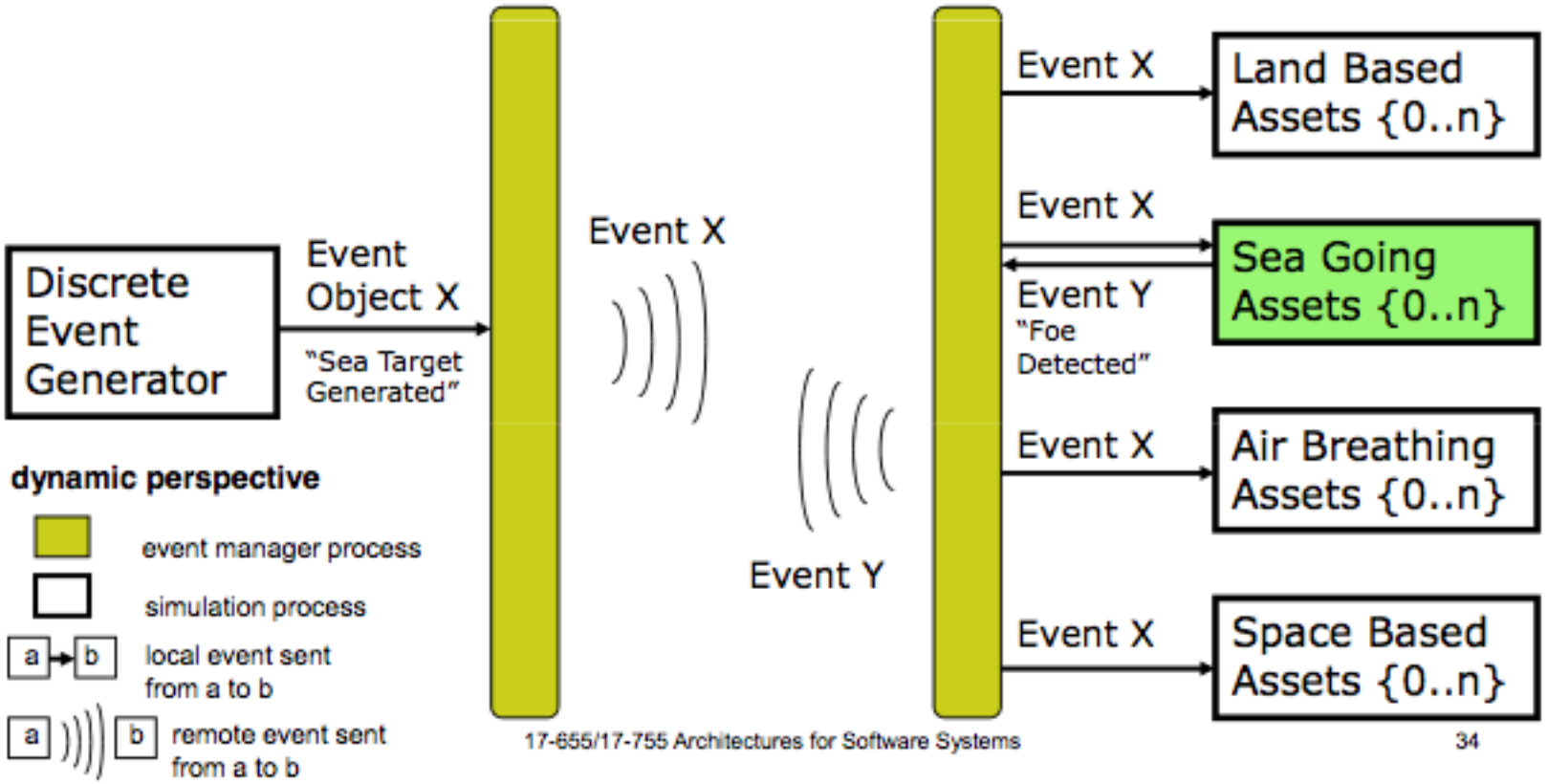
Example 2 – Explicit Invocation



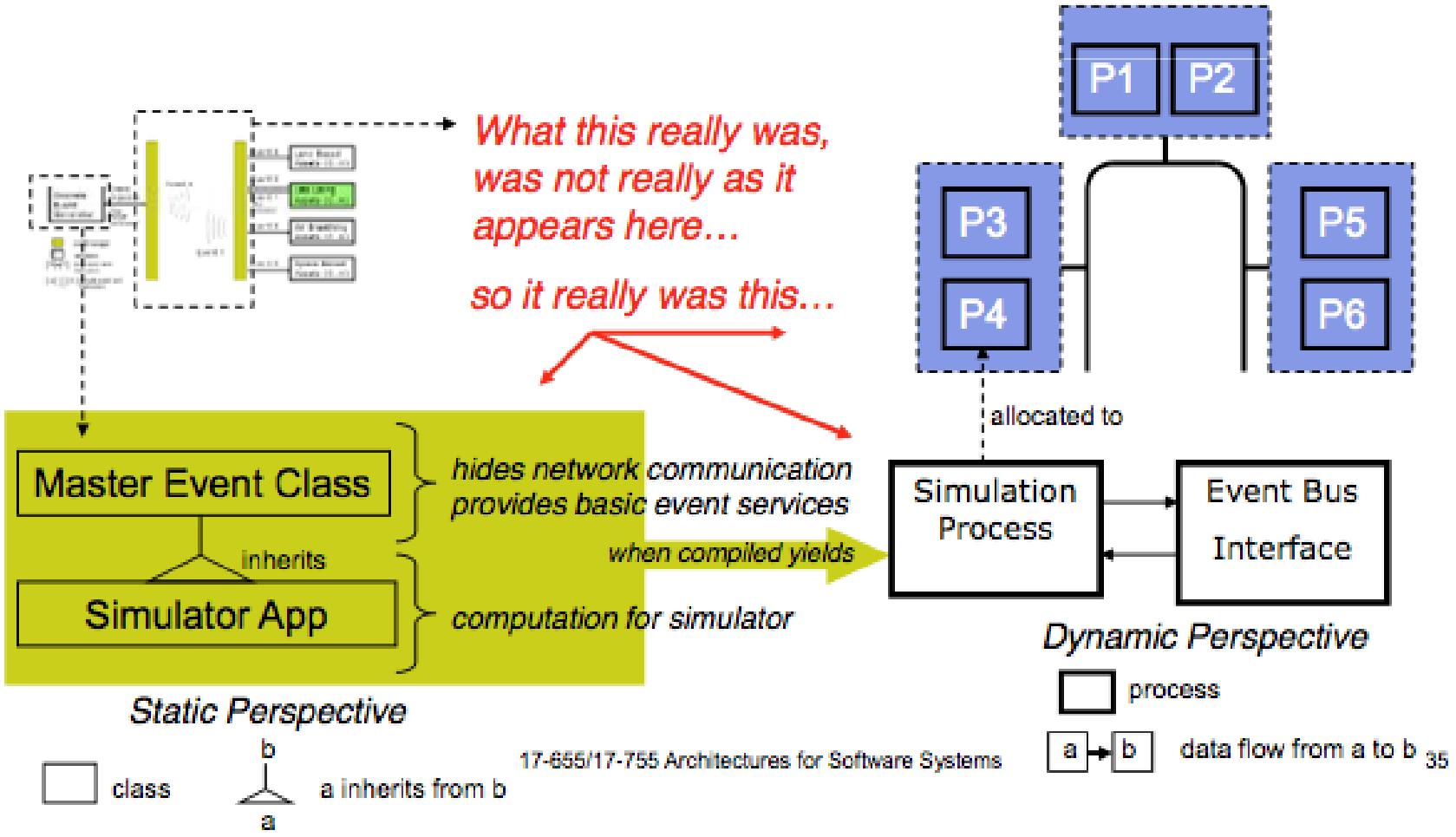
Publisher-Subscriber 패턴

- 프로그래밍 언어
  - C 언어로 구현하기 어려움
  - 자바 언어를 사용할 경우 Observer, Observable을 사용하여 구현할 수 있음

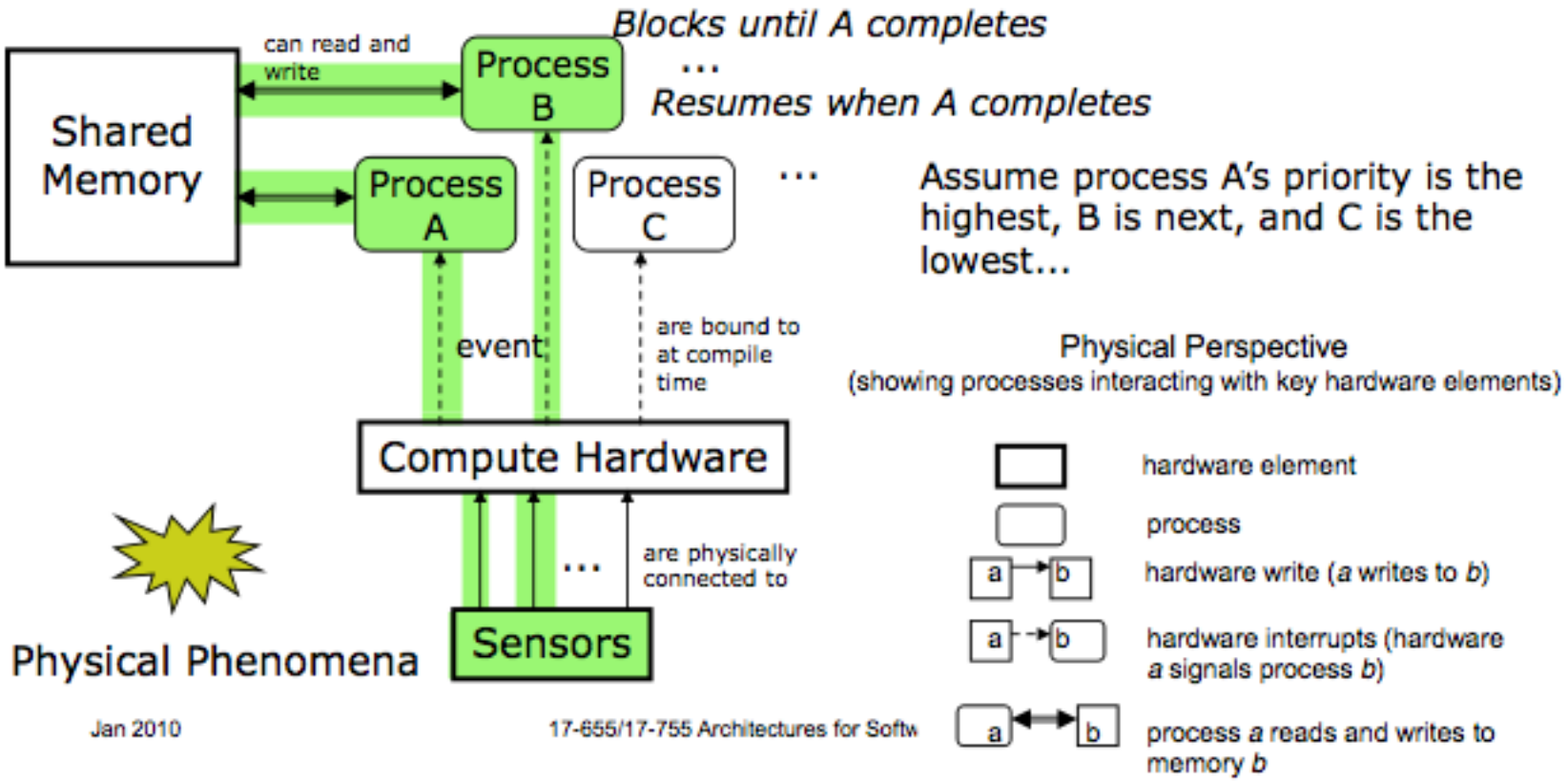
Publisher-Subscriber 패턴



Publisher-Subscriber 패턴



Publisher-Subscriber 패턴



Publisher-Subscriber 패턴

❖ 장점
- 다수의 컴포넌트에게 동시에 변경 공지를 할 수 있음
- GUI 인터페이스를 쉽게 만들 수 있음
- GUI 빌더나 프레임워크를 쉽게 만들 수 있음

❖ 단점
- Non-deterministic 한 문제
- 이벤트 핸들러와 프로세스 로직이 Tightly coupled 되어 있다.

Questions?