Software Architecture

Implicit Asynchronous Communication Software Architecture

버퍼가 없는 문제. 이벤트가 나오면 빨리 핸들링 해줄 수 있음.

Non-buffered Event-Based Implicit Invocations Architecture Style

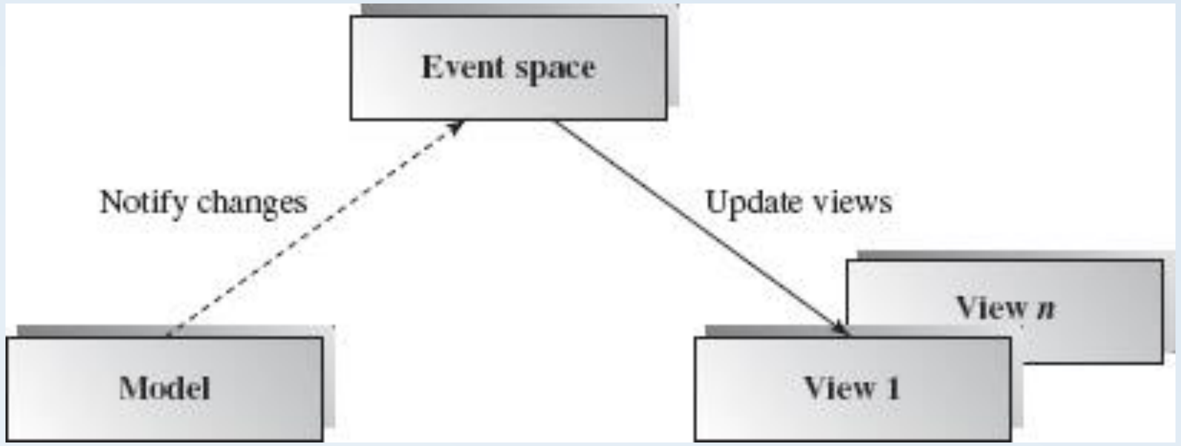
Eunmi Choi
Kookmin University

Non-buffered Event-Based Implicit Invocations

- Structure
  - Breaks the software system into two partitions:
    - event sources and event listeners.

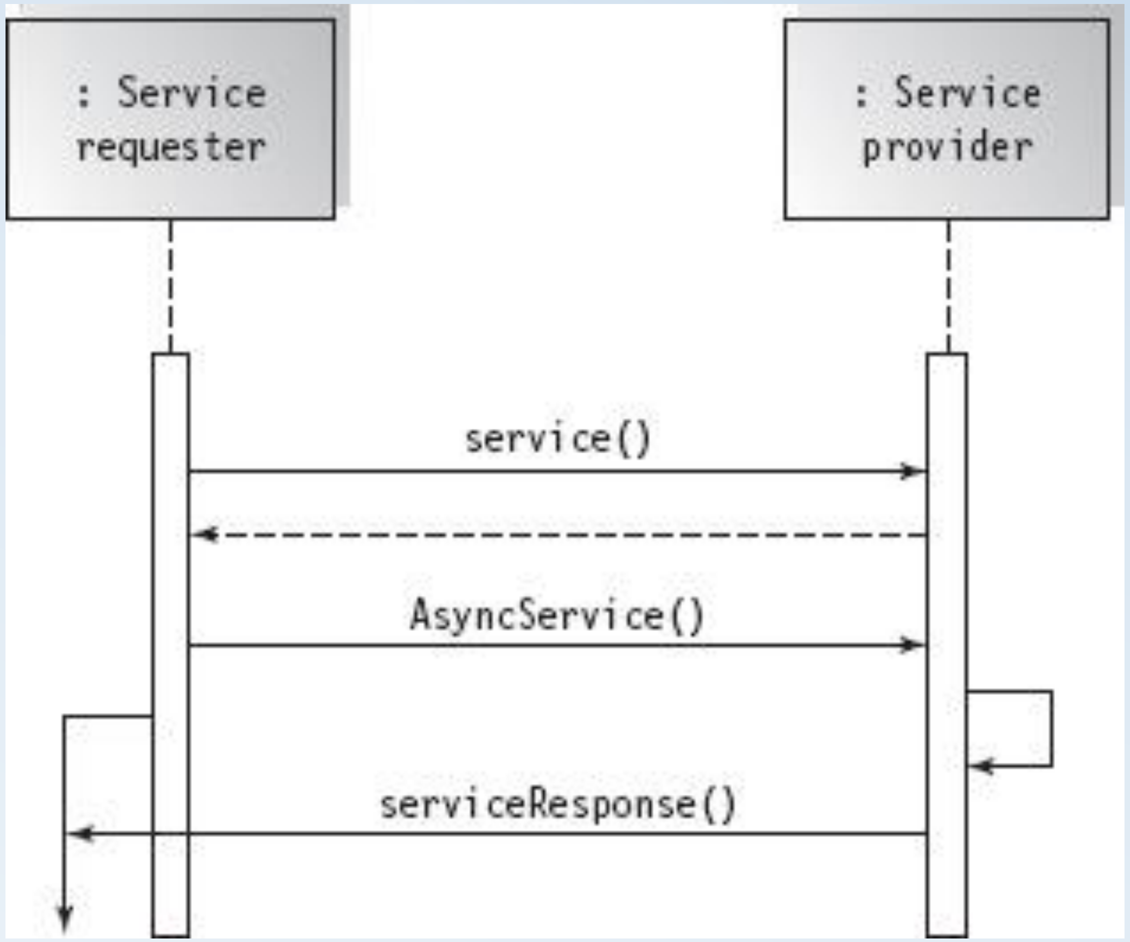
이벤트소스/리스너가 있음.
이벤트 핸들러는 버퍼가 있지만 소프트웨어 아키텍트가 버퍼를 설계할 필요는 없음.

  - The event registration process connects these two partitions
  - There is no buffer available between these two parties.
- Ex) Event implicit invocations in SmallTalk



Non-buffered Event-Based Implicit Invocations

- Synchronous vs. asynchronous invocations



기억할 것!!!
가로 화살표가 아래가 아니라 위로 연결된 경우도 있음.
어느 레벨에서 연결하겠다

질문했으니 마지막에 들을것 - 5/7 17분쯤부터

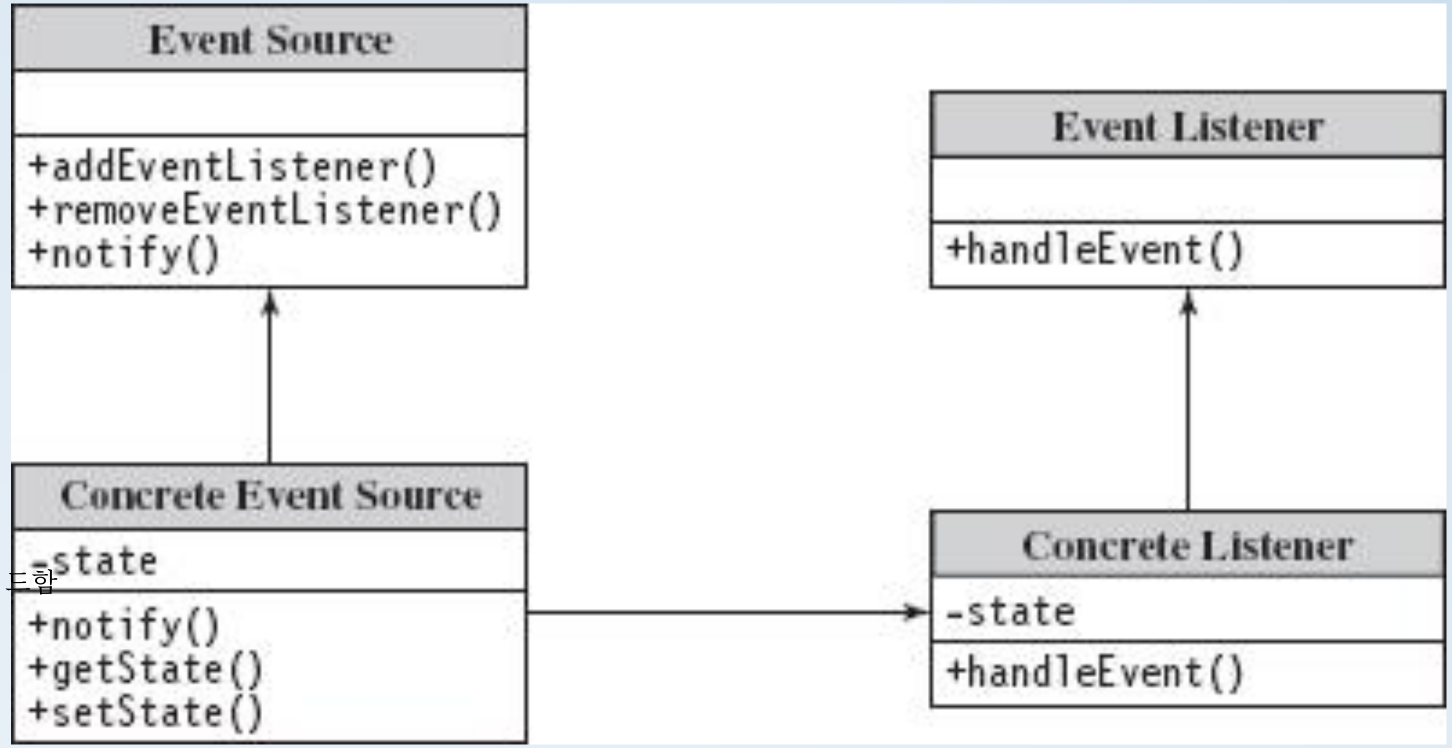
일반적/세부적/타입/폴리모피즘

Non-buffered Event-Based: Example

- Class diagram for event-based implicit invocation architecture

클래스 단계의 다이어그램으로 나옴.

수퍼클래스



짹짹에 대한 핸들링

노티파이를 오버라이드함

개골하면 개굴에 대한 핸들링

Examples in Java

```
package myEvent;
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Source extends Applet {

  private TextField text1;
  private Button button1;
  private Target target;

  // Source is the place where event is
triggered

  public void init() {
     text1 = new TextField(20);
     add(text1);

     button1=new Button("Click Me");
     add(button1);      개골

     target = new Target(button1,text1);
     button1.addActionListener(target);
  }
}
```

```
package myEvent;
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Target implements ActionListener {

  private Button button1;
  private TextField text1;

  public Target(Button b, TextField t) {
    button1 = b;
    text1 = t;
  }
 // Target is the event listener                    개골에 대한 액션
  public void actionPerformed (ActionEvent event) {
    String msg = new String("Hello");

     if (event.getSource()==button1) {
       text1.setText(msg + " " + text1.getText());
     }
   }
}
```

Examples in Java

```
public interface MyListener extends EventListener
{ void handleEvent(EventObject e);
}

public class MyEventObject extends EventObject {
  long t1;
  public MyEventObject(Object o) {
   super(o);
   t1 = System.currentTime();
  }
  public long getTime()  { return t1; }
}

public class Sink implements MyListener {
  public void handleEvent(EventObject e) {
   System.out.println("Time is" + e.getTime());
  }
}
```

컨슘

```
public class Source implements Runnable {
  Vector v=new Vector();
  Thread thread;

  public Source(){
    Thread = new Thread(this);
    thread.start();
  }
   public void run() {
    while (true) {
     triggerEvent();
     try {
      thread.sleep(1000);
     } catch (Exception e) {. . .}
  }
  public synchronized void addMyListener (MyListener l)
  {
    v.addElement (l);
  }
  void triggerEvent() {
    MyEventObject meo = new MyEventObject (this);
    for (int i = 0; i < v.length; i++) {
     MyListener wl=v.elementAt(i);
     wl.handleEvent(meo);
    }
  }
}
```

스레드가 됨.
... 아 몰겠당...

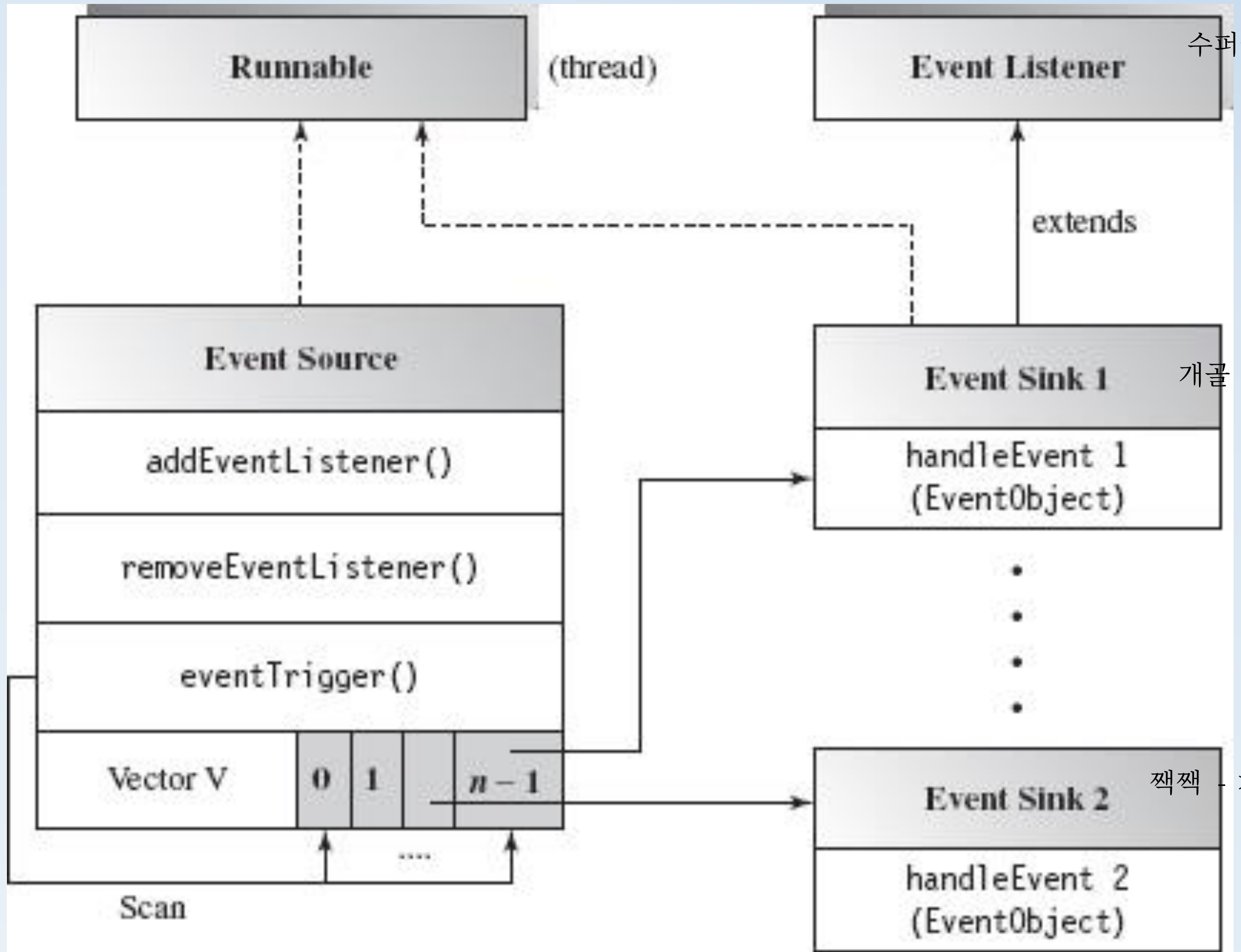
이 코드는 스레드로
핸들링할 때

Examples in Java

```
public static void main(String[] args) {
  Source s1 = new Source();
  Sink t1 = new Sink();
  s1.addMyListener(t1);
   .    .
}
```

Non-buffered Event-Based: Example

- Class diagram for user's defined event source and event sinks
  - user-defined event sources and event listeners (targets) that may be running in different threads concurrently.



수퍼클래스

만드는 것은 프레임워크에서 제공
라이브러리에서 인터페이스를 사용할
수 있도록 함.

개골 - > 에대한 핸들링 구조

tm레드가있어서 어떤
일들이 진행됨

짹짹 - > 에 대한 핸들링 구조

Non-buffered Event-Based: Application

이벤트로 처리하ㅡㄴㄴ 아키텍쳐 스타일은 어플리케이션에서 많이 사용 - 이벤트에 대한 트리거 개념
이런 핸들링 디버깅 많은 ide, 어플리케이션 쪽에서 씀

- Application Examples
  - The trigger function in the Oracle Developer database application
  - Many IDE for CASE tools
    - The completion of the editing of a source file may trigger its compilation, or a new recompilation; this in turn triggers the relink processing, and so on.
  - Handling debugging
    - stopping at a breaking point may trigger the editor to take over so that a tester can check the current status of the program.
  - Stock managing system

    어플리케이션 - 주식에 대한 것. 어퍼와 로우 한계점을 두어서 주식이 너무 내려갈 때 너무 올라갈 때에 대한 이벤트를 걸어두면 트리거 됨. 이벤트베이스의 프로듀스 - 컨슈머

    - a user can set upper and lower threshold levels of a specific stock. When the stock value goes either too high or too low, the user will be notified automatically by the system and action will be taken automatically which can be configured by the user in advance
  - E-commerce system

    재고관리에 대한 어플리케이션 - 재고가 1개 남았을 때 주문을 해야겠다! 하는 최소 단계의 스탑 레벨을 걸어놓고 내려가면 오더를 하도록 해둠. 항상 재고가 유지되면서 판매를 이어ㅏ낙ㄹ 수 있음. 뭔가 액션을 맞춰 빨리 해주는 경우에 이벤트 베이스로 함.

    - when the inventory level of an item goes below the minimum stock level, the system is set to notify the contracted suppliers to order more items in order to keep the item in the stock

Non-buffered Event-Based: Application

- Applicable domains
  - Interactive GUI component communication and integrated development environment (IDE) tools

    gui 쪽 많이 사용
  - Applications that require loose coupling between components that need to notify or trigger other components to take actions upon asynchronous notifications
  - The implementation of state machines
  - When event handlings in the application are not predictable

이벤트 핸들링

Non-buffered Event-Based: Benefits

- **Benefits:**
  - *Framework availability:* Many vendor APIs such as Java AWT and Swing components are available.

    프레임워크에서 주로 사용함. 버퍼가 없는 이벤트) awt나 swing, gui에서 쓰는걸 그대로 사용함.
  - *Reusability of components:* It is easy to plug in new event handlers without affecting the rest of the system.

    핸들러를 만들러 놓으면 다른 액션에 넣을 수 있음. 재사용 가능
  - *System maintenance and evolution:* Both event sources and targets are easy to update.

    이벤트를 만들고 컨숨(소스 - 타겟)이 세파레이트 컨숨이 일어남. 컨수머만, 핸들러만, 이벤트가 일어나는 것만 생각하면 됨. 서로 분리되는 좋은 구조를 가짐.
  - *Independency and flexible connectivity:* Dynamic registration and deregistration can be done dynamically at runtime.

    독립적이고 유연한 연결성. 핸들러를 서로 add해놓고 임플리메이션과 업데이트 시 무관하게 관리할 수 있음.
  - Parallel execution of event handlings is possible.

    여러핸들러가 있으면 패러렐한 익스큐션이 가능
    유사한 문제를 한순간 같이. 같은 인스트럭션 다른 데이터.

Non-buffered Event-Based: Limitations

얘가 왜 깨어났는지 모름. 디버깅, 테스팅이 어려움. 어떻게 트리거 됐는지 다 봐야하는 것.
세파레이트 콘설ㄴ 이긴 한데, 얘가 상황에 따라 꼭 얘만 써야 하는 문제가 있음. 커플링이 있음
간접적으로 부를 때 오버헤드가 일어나고 신뢰도도 떨어짐. 테스팅 디버깅이 어려우니.

- **Limitations:**
  - It is difficult to test and debug the system since it is hard to predict and verify responses and the order of responses from the listeners. The event trigger cannot determine when a response has finished or the sequence of all responses.
  - There is tighter coupling between event sources and their listeners than in message queue-based or message topic-based implicit invocation.
    - Data sharing and data passing in the event object forwarded from event sources to event listeners also make the coupling tighter and somewhat hard to debug and test.
  - Reliability and overhead of indirect invocations may be an issue.
- **Related architecture:**
  - PAC, message-based, MVC, multi-tier, and state machine architectures