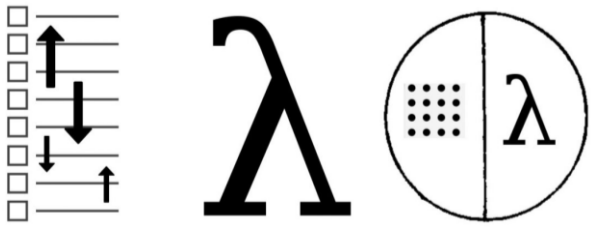
Fundusze Europejskie
Wiedza Edukacja Rozwój

Politechnika Wrocławska

Unia Europejska
Europejski Fundusz Społeczny

„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”



Programming paradigms

L14: Event-driven programming

by Michał Szczepanik

Event-driven programming

Event-driven programming is a programming paradigm in which the flow of program execution is determined by events - for example a user action such as a key press, mouse click or a message from the operating system or any another program.

An event-driven application is designed to detect events as they occur, and then deal with them using an appropriate event-handling procedure.

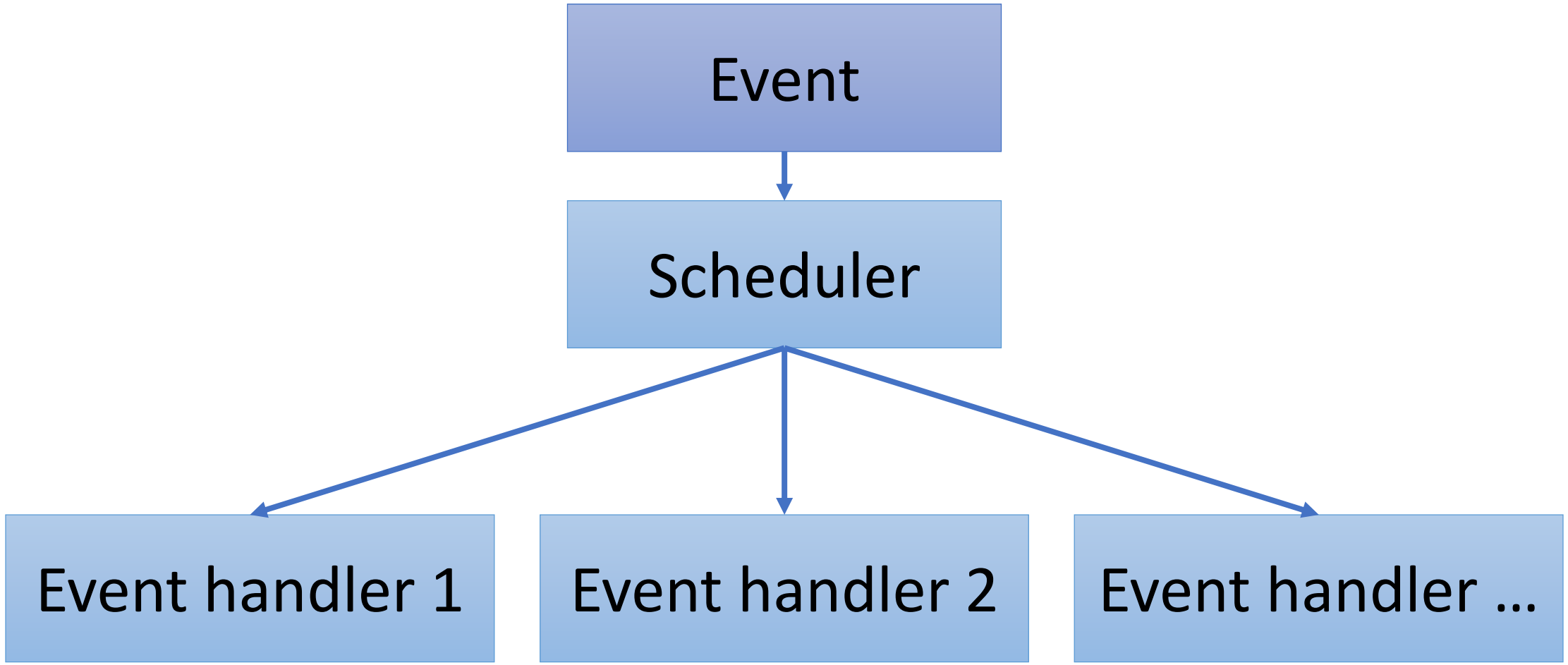
History

The change in emphasis from procedural to event-driven programming has been accelerated by the introduction of the *Graphical User Interface* (GUI) which has been widely adopted for use in operating systems and end-user applications.

Scheduler

The central element of an event-driven application is a scheduler that receives a stream of events and passes each event to the relevant event-handler. The scheduler will continue to remain active until it encounters an event (e.g. "End_Program") that causes it to terminate the application. Under certain circumstances, the scheduler may encounter an event for which it cannot assign an appropriate event handler. Depending on the nature of the event, the scheduler can either ignore it or raise an *exception* (this is sometimes referred to as "throwing" an exception).

Scheduler


Event
Scheduler
Event handler 1
Event handler 2
Event handler …

Important terms

Event Object

Everything in Java is object-oriented. It stands to reason then, that even events are tied to objects. Basically, the event itself is an object, thus we refer to this general concept as the event object.

Event Source

Knowing that an event object exists doesn't help unless you know what the element firing the event object is. For example, is it a button, a checkbox, or a drop-down menu? The event source is object that is triggered in the event.

Event Listener

In Java GUI applications, we need to specifically identify components that should react to events. A button, for example, is just a button unless you tell Java that you want to actually do something when it is clicked. In other words, we need to make sure the button listens for an event.

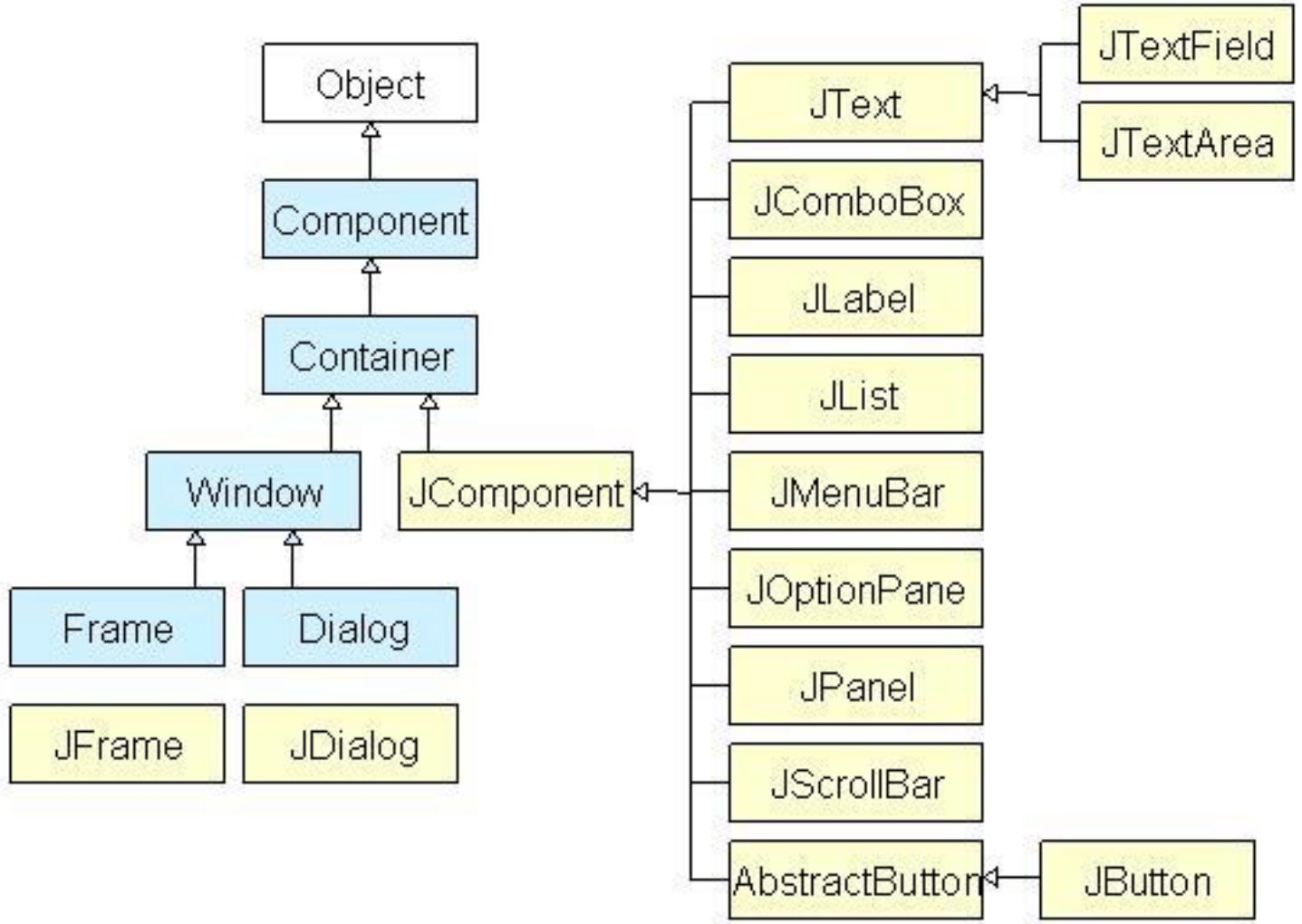
GUI: Java Swing

Java Swing is a lightweight Graphical User Interface (GUI) toolkit that includes a rich set of widgets.

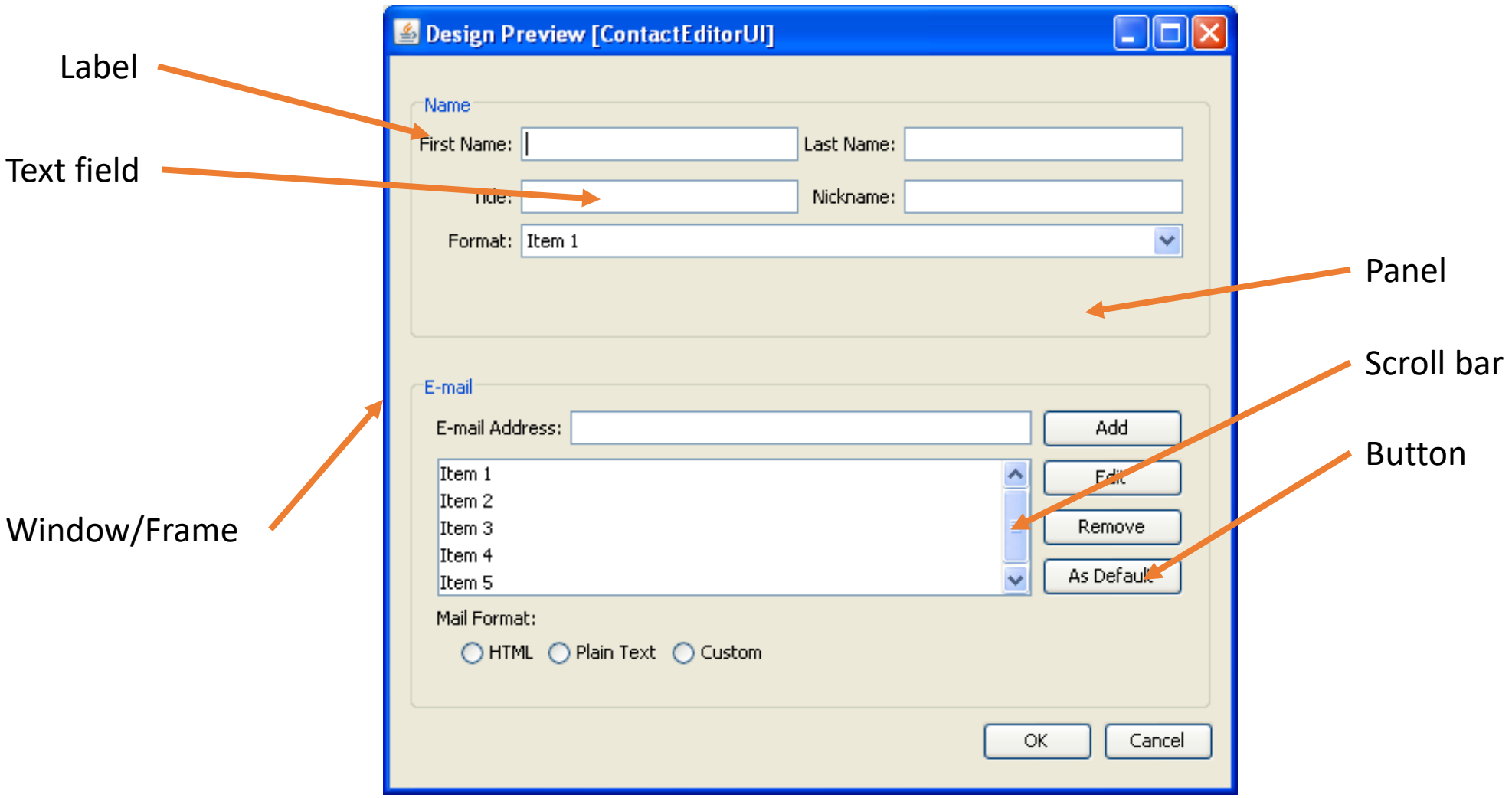
It includes package which allow to make GUI components for Java applications, and It is platform independent.

The Swing library is built on top of the Java Abstract Widget Toolkit (AWT), an older, platform dependent GUI toolkit.

Java Swing class Hierarchy Diagram



GUI components



Container classes

Container classes are classes that can have other components on it.

There are three types of them:

- **Panel**: It is a pure container and is not a window in itself. The sole purpose of a Panel is to organize the components on to a window.
- **Frame**: It is a fully functioning window with its title and icons.
- **Dialog**: It can be thought of like a pop-up window that pops out when a message has to be displayed. It is not a fully functioning window like the Frame.

Sample Swing GUI

```
import javax.swing.*;
class gui{
  public static void main(String args[]){
    JFrame frame = new JFrame("My First GUI");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300,300);
    JButton button1 = new JButton("Button 1");
    JButton button2 = new JButton("Button 2");
    frame.getContentPane().add(button1);
    frame.getContentPane().add(button2);
    frame.setVisible(true);
  }
}
```

Action Listener

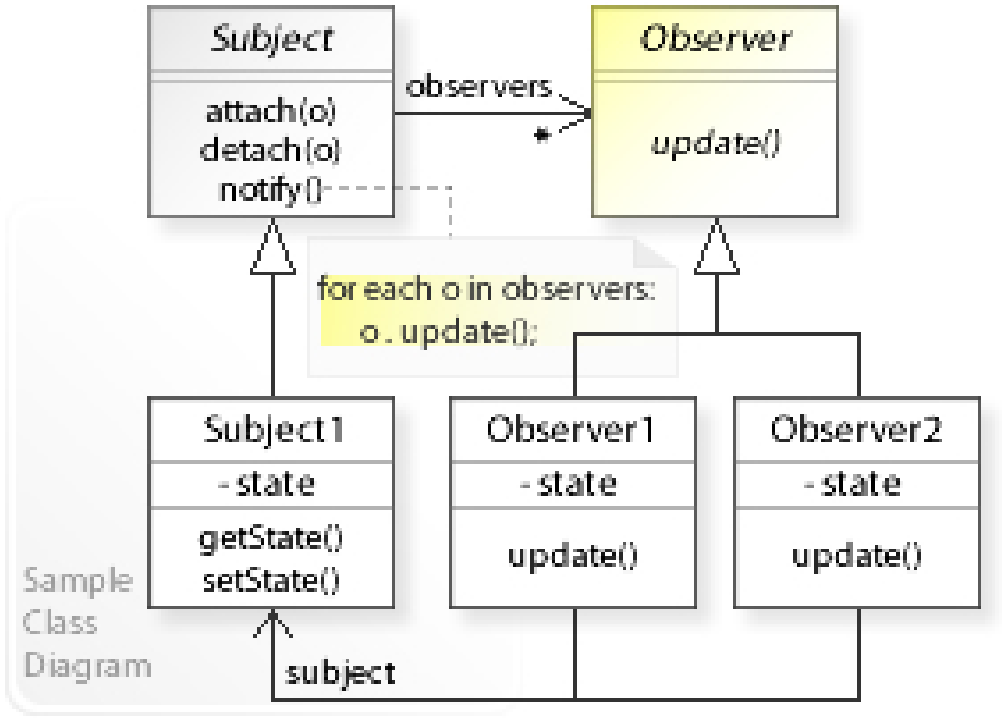
```
…
button1.addActionListener(new ActionListener(){
  public void actionPerformed(ActionEvent e){
    tf.setText("Welcome to PP.");
  }
});
…
```

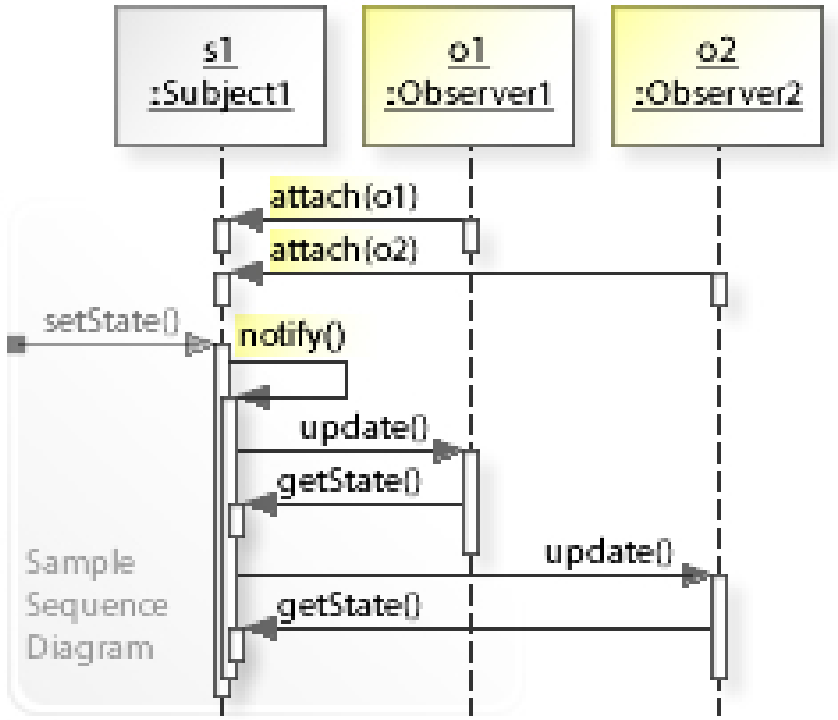
Observer pattern

The Observer pattern addresses the following problems:

- A one-to-many dependency between objects should be defined without making the objects tightly coupled.
- It should be ensured that when one object changes state an open-ended number of dependent objects are updated automatically.
- It should be possible that one object can notify an open-ended number of other objects.

Observer pattern





Implementation

Observer pattern uses three actor classes:

- Subject,
- Observer,
- Client.

Subject is an object having methods to attach and detach observers to a client object. Programmer have to created an abstract class Observer and a concrete class Subject that is extending class Observer.

Check list

- ✓Differentiate between the core (or independent) functionality and the optional (or dependent) functionality.
- ✓Model the independent functionality with a "subject" abstraction.
- ✓Model the dependent functionality with an "observer" hierarchy.
- ✓The Subject is coupled only to the Observer base class.
- ✓The client configures the number and type of Observers.
- ✓Observers register themselves with the Subject.
- ✓The Subject broadcasts events to all registered Observers.
- ✓The Subject may "push" information at the Observers, or, the Observers may "pull" the information they need from the Subject.

Publish/subscribe communication model

Java GUIs support event-driven programming according to the publish/subscribe model.

- Components are publishers of events.
- Other objects implement event-handling code specific to a certain event.
- A handler for an event E subscribes (that is, registers) to a component that publishes events E.
- Whenever a component triggers an event E, it notifies all handlers for that event that have been registered.
- A notified handler executes its code in response to the event.
- Eventually, control returns to the component.

Publish/subscribe event-handling in Java

Java GUIs support event-driven programming according to the publish/subscribe model.

- Swing components include methods of the form addEListener(EListener handler) to register handlers of a specific event E
- Type EListener corresponds to an interface, whose methods are called whenever the corresponding events are triggered
- A handler for event E is an object of a subtype of EListener; that is, the handler's class implements interface E
- Whenever a component triggers an event E, it notifies all handlers for that event that have been registered by calling the proper method of EListener

In general, handlers and publishers are completely independent. In practice, they often need to communicate and share state, because "handling" an event often requires to change some parts of the GUI

Swing event listerers

Listener	Description
ActionListener	Responds to actions (e.g., button click, text field change)
ItemListener	Listens for individual item changes (e.g., a checkbox)
KeyListener	Listens for keyboard interaction
MouseListener	Listens for mouse events (double-click, right-click, etc.)
MouseMotionListener	Listens for mouse motion
WindowListener	Acts on events in the window
ContainerListener	Listens for container (JFrame, JPanel) events
ComponentListener	Listens for changes to components (e.g., a label being moved, hidden, shown, or resized in the program)
AdjustmentListener	Listens for adjustments (e.g., a scroll bar being engaged)

Heavy computations

Allocate heavy computations to a special executor (thread) in the Java system, which can run the computation in the background without blocking the GUI.

```
SwingWorker worker = new SwingWorker<Void, Void>() {
  @Override
  public Void doInBackground() {
    // encode 2-hour video in high resolution
    return null;
  }
};
```

Start the worker's computation: worker.execute()

Check whether the worker has finished computing: worker.isDone() (returns boolean)

Get the results of the worker's computation after completion: worker.get()

Thank you for your attention

michal.szczepanik@pwr.edu.pl