

# Software Architecture

## Lecture 4 Event Systems

João Pedro Sousa  
George Mason University

---

### previously data flow and call-return styles

#### data flow

batch sequential  
dataflow network (pipe & filter)  
acyclic, fan-out, pipeline, Unix  
closed loop control

#### call-and-return

main program/subroutines  
information hiding - objects  
stateless client-server  
SOA

#### interacting processes

communicating processes  
event systems  
implicit invocation  
publish-subscribe

#### data-oriented repository

transactional databases  
stateful client-server  
blackboard  
modern compiler

#### data-sharing

compound documents  
hypertext  
Fortran COMMON  
LW processes

#### hierarchical

tiers  
interpreter  
N-tiered client-server

## today event-based styles

### data flow

batch sequential  
dataflow network (pipe & filter)  
acyclic, fan-out, pipeline, Unix  
closed loop control

### call-and-return

main program/subroutines  
information hiding - objects  
stateless client-server  
SOA

### interacting processes

communicating processes

#### event systems

implicit invocation  
publish-subscribe

### data-oriented repository

transactional databases  
stateful client-server  
blackboard  
modern compiler

### data-sharing

compound documents  
hypertext  
Fortran COMMON  
LW processes

### hierarchical

tiers  
interpreter  
N-tiered client-server

## today's outline

- interacting processes style
  - family tree:
    - communicating peers
    - publish-subscribe
    - implicit invocation
- event systems
  - QAs
  - implementation
  - Lab 2

#### Acknowledgment

some of the material presented in this course is adapted from 17655,  
taught to the MSE at CMU by David Garlan and Tony Lattanze

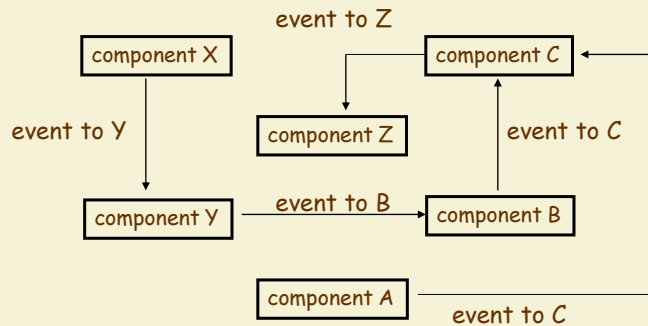
## communication is loosely coupled in the interacting processes style

- components
  - independent threads of control
  - implemented as a process or thread
  - may be distributed
- connectors
  - communication is loosely coupled and often asynchronous
- system
  - components may or may not have knowledge of other components
  - functionality of one component does not depend upon others
  - overall system functionality depends upon all components functioning and communicating properly

## interacting processes family tree

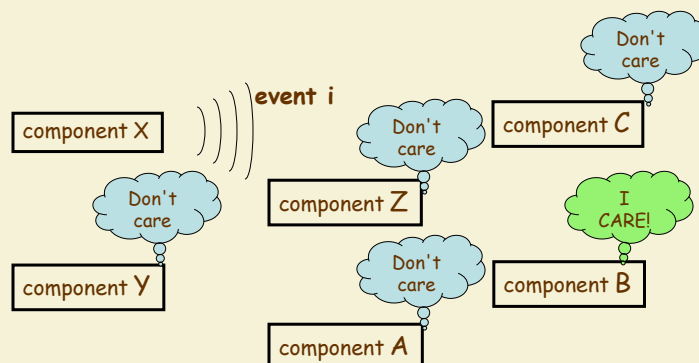
- communicating peers
- event systems aka implicit events
  - publish aka broadcast
  - publish-subscribe
  - implicit invocation

## communicating peers know recipient's identity



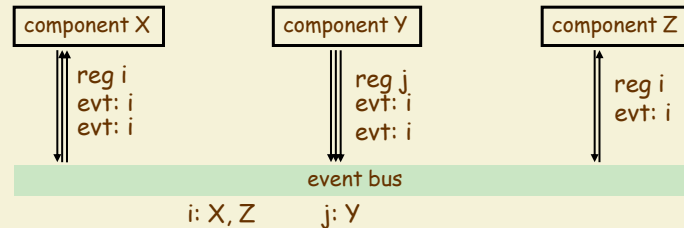
- identity of event recipients is known by event senders
- order of execution can be prescribed/predicted

## event systems publish aka broadcast



- Ethernet protocol in local area networks uses similar idea:  
event packet is tagged with recipient's address

## publish-subscribe & often implicit invocation rely on event infrastructure



- identity of event recipients is unknown to senders
- order of event delivery is bus-dependent
  - different event buses make different guaranties
  - or no guaranties about ordering

## implicit invocation recipients register callback



- directly with event issuer
  - e.g. supported by Java Observable/Observer classes
- facilitated by broker compare w/ publish-subscribe
  - e.g. database triggers; lab 2

## in summary

- **communicating peers**
  - asynchronous messages aka explicit events  
explicit wrt identifying the recipient
- **event systems** aka implicit events
  - events delivered to all interested components in some order
  - **publish** aka broadcast
  - **publish-subscribe**
    - interested components subscribe to events
    - interested components receive asynchronous message
  - **implicit invocation**
    - interested components register a callback method
    - upon the event, the method is invoked (call-return)

## today's outline

- interacting processes style
- **event systems**
  - QAs
  - implementation
    - single process
    - distributed
  - Lab 2

## events are widely supported

- *C* signals: OS (software) interrupts sent by *kill* function program can specify handler function for each signal
- *Ada*: defines *interrupts* and *interrupt handlers*
- *C++*: defines *events*, *event sources* and *event receivers*
- Visual Basic: widgets send various events depending upon user interaction, timers, and so forth; code attached to event is executed when event occurs
- Java *Observer* & *Observable* classes
- database triggers: conditions associated to data trigger callbacks to registered methods
- middleware support for events: CORBA, etc.
- COTS event buses: JMS, IBM MQ Series, etc.

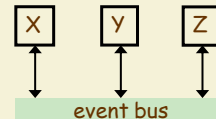
## high-end, large-scale simulation HLA example

- simulation is big business
  - hundreds of simulators, vendors
  - one training system for the Army cost \$2 Billion alone
- problem: combine simulators into a joint "exercise"
  - may involve dozens or hundreds of simulators
  - usually highly distributed
  - produced by multiple vendors
- version 1.0 of the HLA published 1996
  - in 2000 HLA becomes IEEE Standard 1516
- HLA defines event-based standard for simulation
  - protocols for interaction
  - data modeling mechanisms
  - procedures to join and leave *federation* of simulations

## today's outline

- interacting processes style
- event systems
  - QAs
  - implementation
    - single process
    - distributed
  - Lab 2

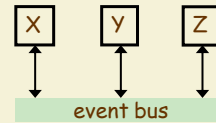
## event systems



- promote scalability
  - easy to add new components
  - however, may increase overhead and hurt performance
- performance may be a challenge
  - hard to coordinate the order of processing
  - predicting and optimizing performance can be a challenge
  - specific performance measurements may still be good

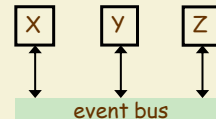


## event systems



- promote reuse
  - decoupling: events offer an interface with little assumptions
  - announcers don't need to know the identity of responders
  - easy to reuse/register a component that communicates this way
- promote conceptual integrity
  - components work more independently
  - interaction policy can be cleanly separated from internals
  - however, due to non-determinism it may be hard to model and reason about run-time behavior

## event systems

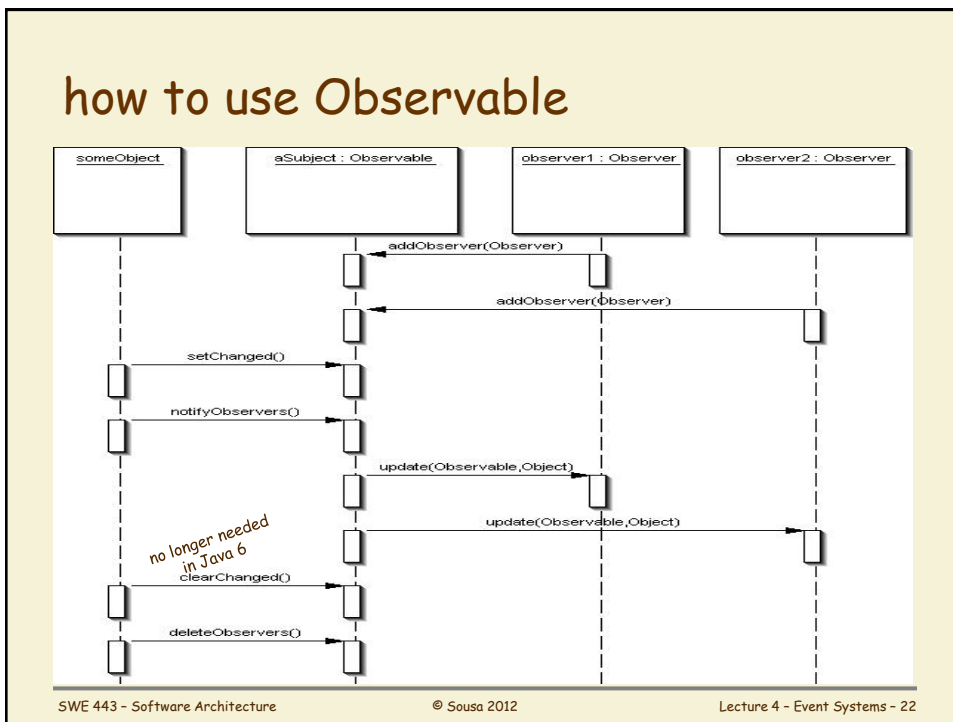
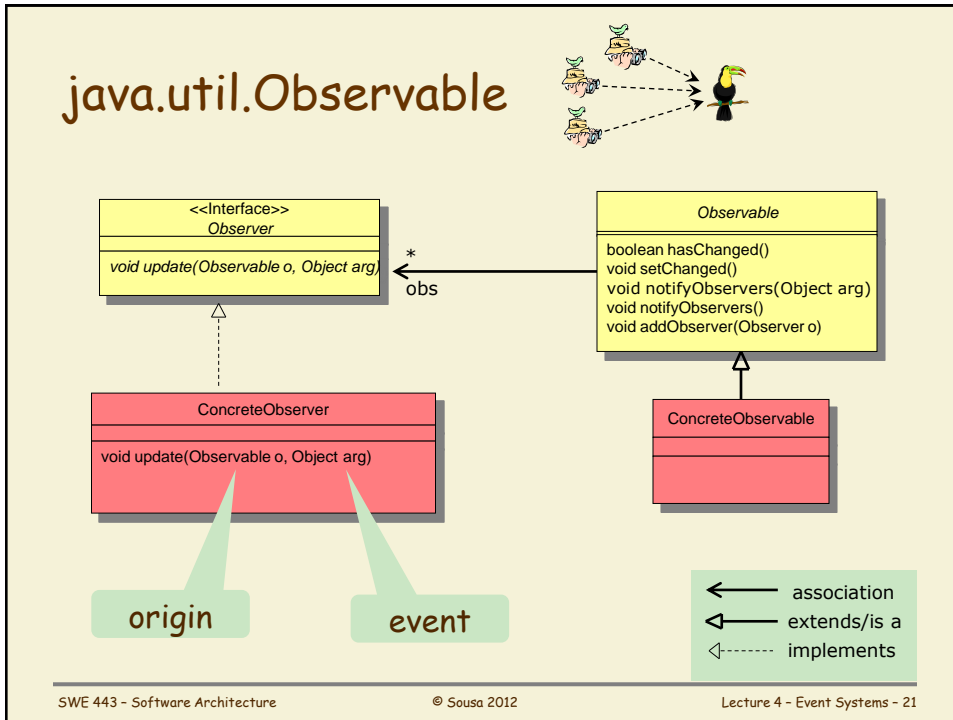


- promote maintainability
  - changes to one component have little or no effect on others
  - however, changes to **when** a component announces/register may impact system behavior
- testing may be a challenge
  - hard to trace origin/path of events
    - may require special tools and/or system structures to do so
  - hard to replicate ordering of actions
    - during testing & after deployment

# take 5

# today's outline

- interacting processes style
- event systems
  - QAs
  - implementation
    - single process: Java Observable/Observer
    - distributed
  - Lab 2



## lab2

### interactive student registration

- as in Lab 1, you are given a working system
- the system maintains
  - list of students: read input file

<i>G number</i>	<i>Last/First Name</i>	<i>Program Affiliation</i>	<i>Account Balance</i>	<i>Course numbers that the student has completed (with no space between the prefix and the number)</i>
G00123456	Carson Kit	ACS	3	CS112 CS211 CS332

- list of courses: read input file

<i>Course Number</i>	<i>Section</i>	<i>Days</i>	<i>Start Time</i>	<i>End Time</i>	<i>Instructor</i>	<i>Course Title</i>
SWE443	A	W	130	420	Sousa	Software Architecture

## lab2

### interactive student registration

- as in Lab 1, you are given a working system
- the system maintains
  - list of students
  - list of courses
- a Student object maintains
  - list of courses taken
  - list of current registrations
- a Course object maintains
  - list of registered students

## lab2

### interactive student registration

- current system provides features
  - list courses
  - list students registered for a course
  - list courses a student has completed
  - ...
- you're asked to enhance the functionality:
  - logging
  - detect course overflow
  - billing
  - refactor checking of course conflicts
- ... and analyze the changes

## lab2

### module view

## lab2

### C&C view

## today's outline

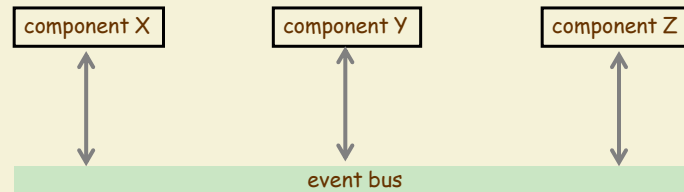
- interacting processes style
- event systems
  - QAs
  - implementation
    - single process
    - **distributed**
  - Lab 2

## distributed event system smart buildings

- control temperature and humidity in a museum  
to preserve delicate paintings
- sensors
  - monitor *environment* and post periodic events  
temperature, smoke, motion, intrusion, humidity...
- controllers
  - monitor events and control *actuators*  
heaters, chillers, humidifiers, dehumidifiers, door locks...
- consoles
  - enable users to set temperature, humidity, enable security...

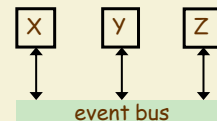
## smart building C&C view

in summary, many event systems  
rely on event infrastructure



- identity of event recipients is unknown to senders
- order of event delivery is unknown
  - different event buses make different guaranties or no guaranties about ordering

in summary, event systems  
easy to modify, hard to test



- QAs promoted due to decoupling and encapsulation
  - reuse
  - modifiability
  - scalability
- QAs inhibited
  - performance: hard to guarantee response time
  - testability: hard to test and reason about correctness
  - availability: possible to miss events (no registrations)

these are general considerations:  
a real analysis requires QA scenarios - next class