11 System Design (1)

Introduction to OOA OOD and UML

2022 Spring

College of Information Science and Engineering

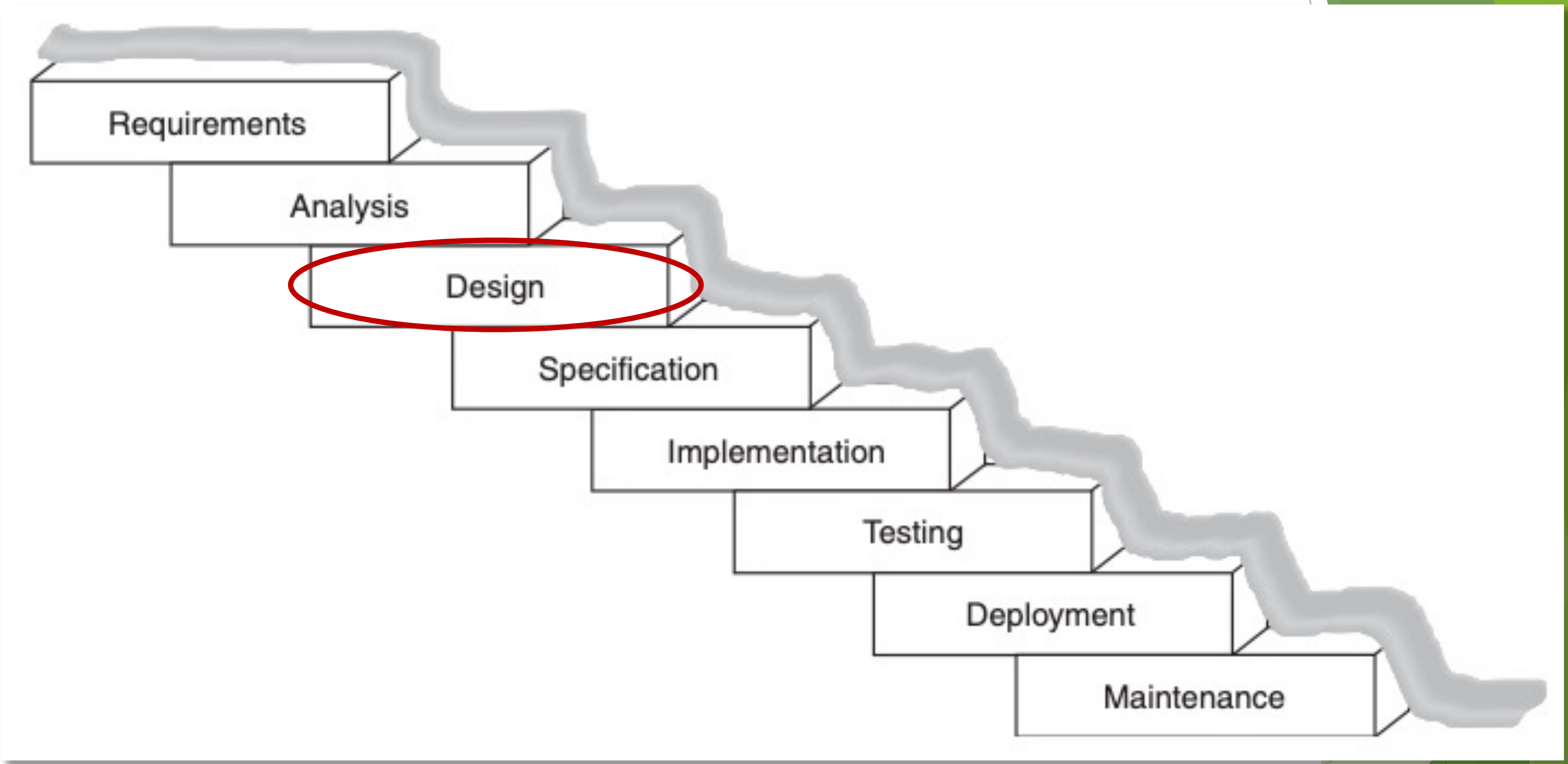
Ritsumeikan University

Yu YAN

Outline

- Introduction
- Choosing a System Topology
- Designing for Concurrency
- Designing for Security Policy
- Partintioning Software
- Exercise 11

Where Are We Now?



- Analysis considers "problem understanding", while Design considers "problem solutions"
- During the design phase, we make certain technology choices (for example, programming languages, protocols and database management systems, etc.)
  - We must decide how much impact we want these choices to have on our design.
- In practice, we seek a system architecture that will support a practical, efficient solution for all the use cases
  - Within that architecture, we perform detailed design for the most important use cases and partial design for the less important ones
  - Between increments, we adjust the priorities, the urgencies and the design, as appropriate

Steps in System Design (1)

- Design can be thought of as having two distinct activities:
  - System design and subsystem design
  - System design forces us to take a high-level view of the task ahead before we proceed with the detail of subsystem design
- System design includes the following activities:
  - Choosing a system topology: how the hardware and processes will be distributed, perhaps over a network
  - Making technology choices: selecting programming languages, databases, protocols and so on; some decisions may be deferred until later in the design phase
  - Designing a concurrency policy: concurrency means many things happening at once – multiple processes, users, machines; these must be coordinated by our software in order to avoid chaos.

Steps in System Design (2)

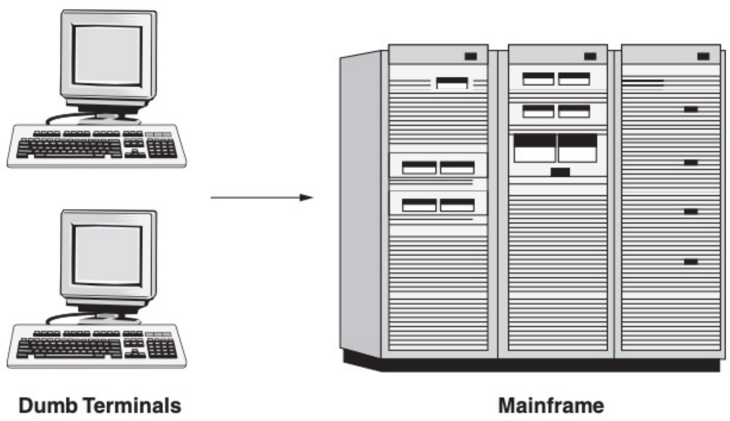
- Designing a security policy: security has a number of aspects, each of which must be properly addressed and controlled
- Choosing subsystem partitions: often, it is impractical to produce a single piece of software that solves all of our problems; instead, we need to produce separate pieces of software and then make sure that the pieces communicate effectively
- Partitioning the subsystems into layers or other subsystems: typically, each subsystem will need to be decomposed further into manageable chunks before we can do detailed design
- Deciding how machines, subsystems and layers will communicate: communication decisions usually happen as a side-effect of the other steps

Outline

- Introduction
- Choosing a System Topology
- Designing for Concurrency
- Designing for Security Policy
- Partintioning Software
- Exercise 11

Types of Topologies (1)

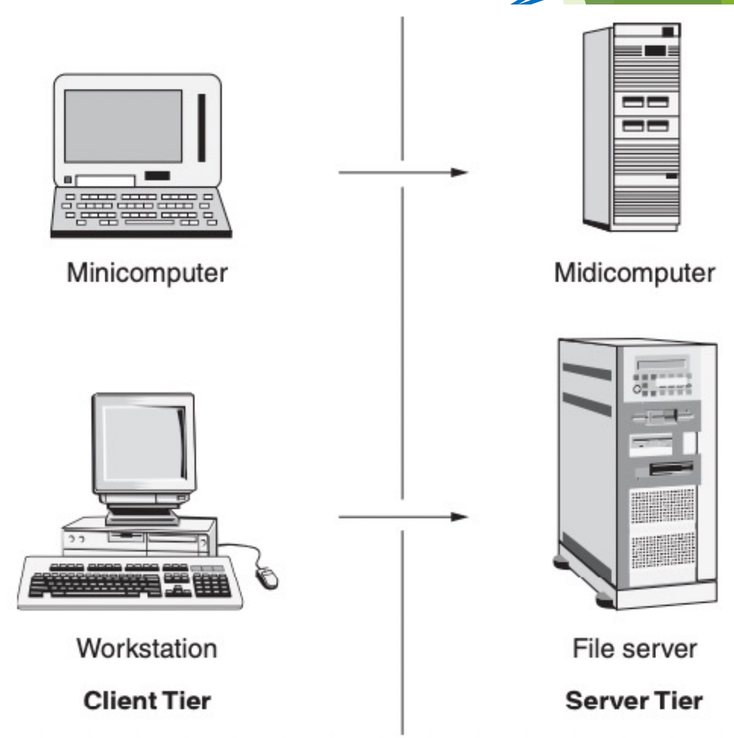
- One-tier architecture: for any given program, there is only one level of computing activity running on one machine



1940s

- Two-tier architecture: Have processing power on each client, with an optional hard drive, so that large central machines would not have to do all the processing.

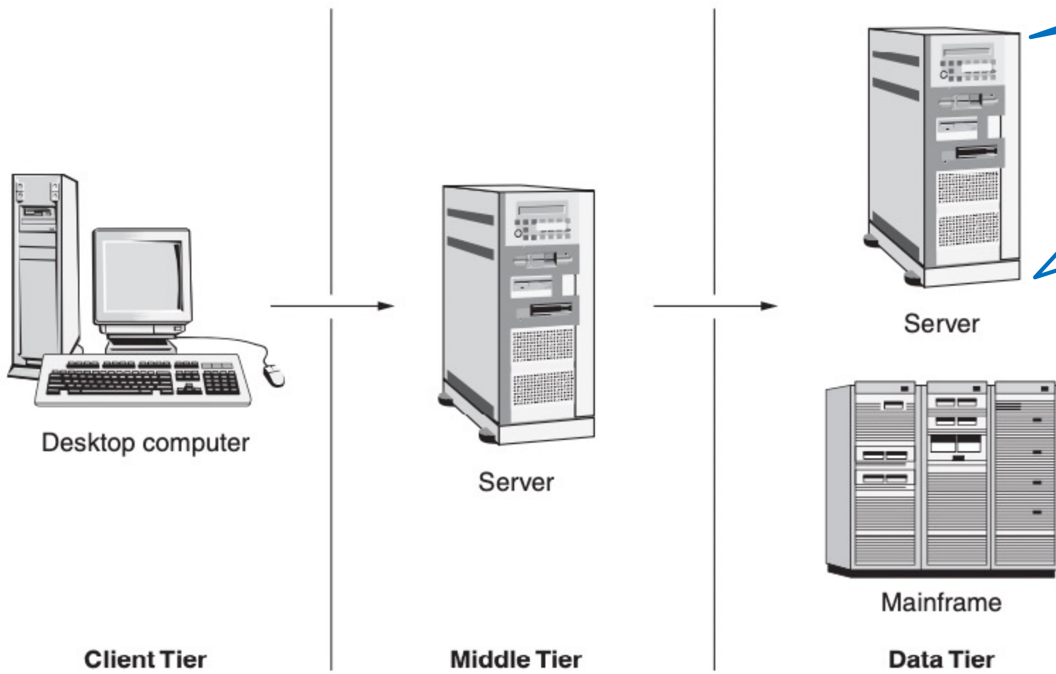
1970s



Types of Topologies (2)

- Three-tier architecture: any one program involves at least three machines:

Since 1990s



Stores the data and provides safe concurrent access to it, DBMS

Presents the user interface to the user, so that they can enter data and view results

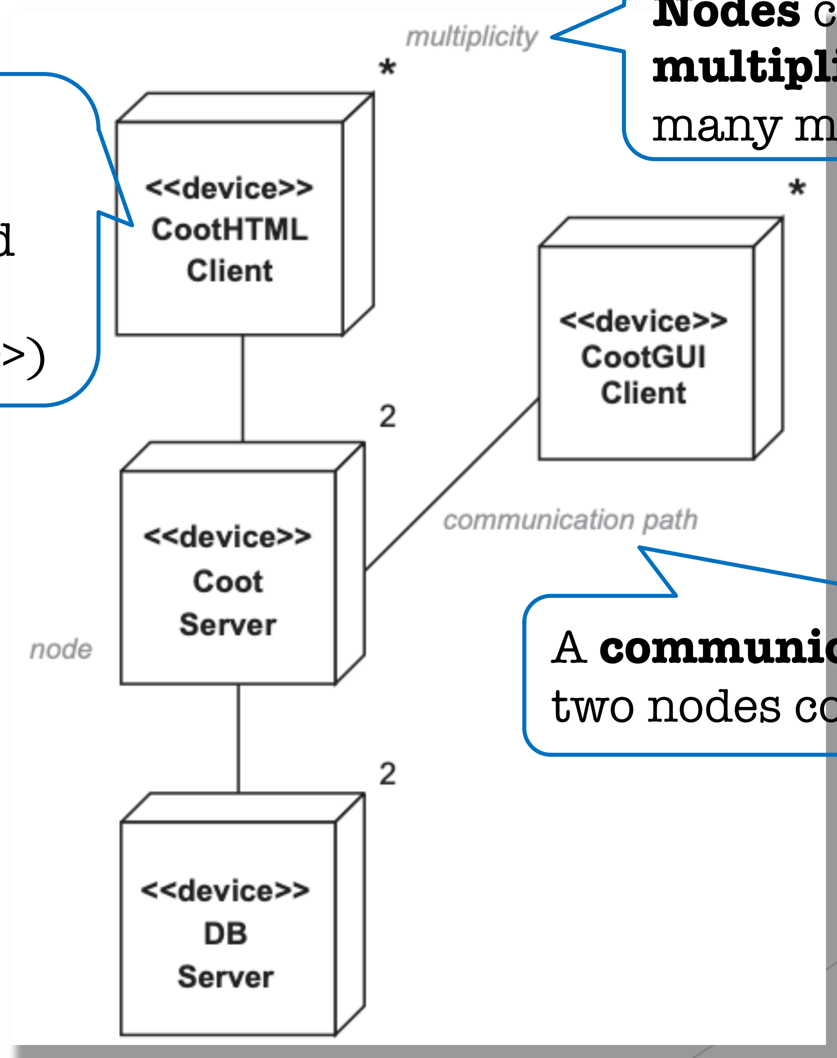
Runs multi-threaded program code using large processors and lots of memory

Depicting Network Topology in UML (1)

- System architectures can be depicted in UML on a deployment diagram

Each **node** represents a host machine (indicated with the UML keyword <<device>>)

Nodes can be given **multiplicities** to indicate how many might exist at run time

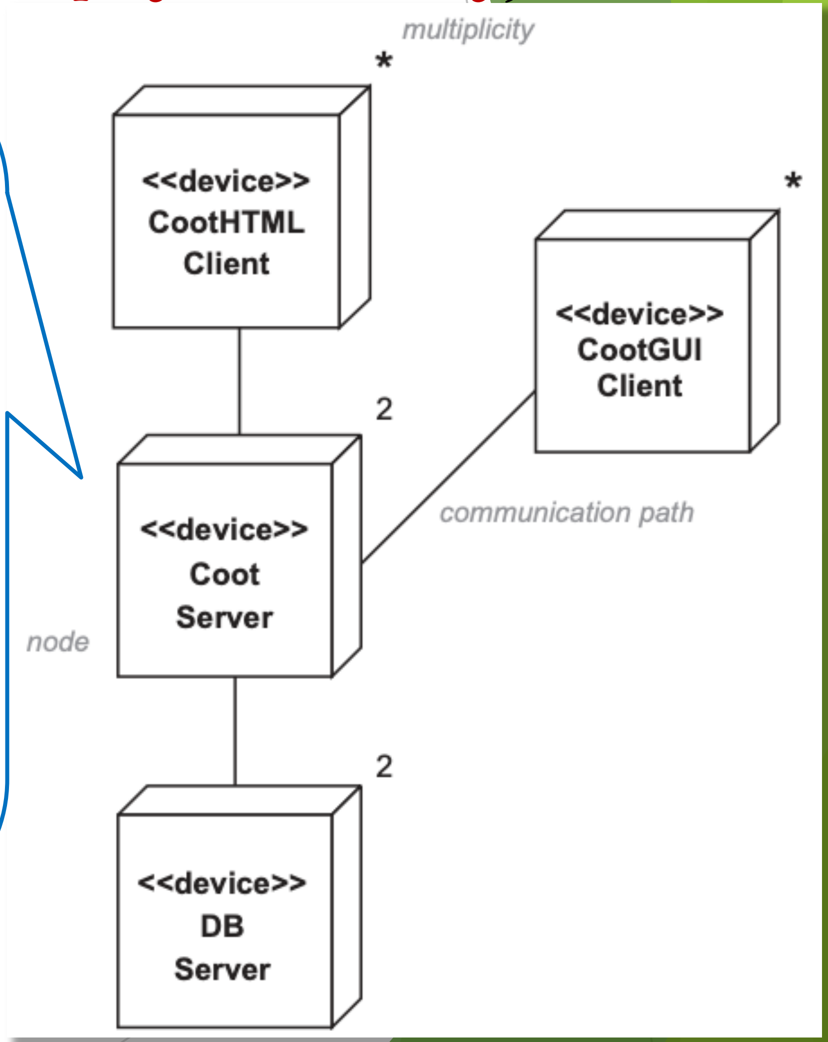


A **communication path** shows that two nodes communicate in some way

Depicting Network Topology in UML (2)

- Most deployment diagrams need an accompanying description if they're to make any sense (we call it a deployment survey)

- The iCoot data tier comprises two database servers (DBServer). Having two such nodes improves throughput and reliability.
- The middle tier, which communicates with the data tier, consists of two server machines (CootServer), again duplicated for the sake of reliability and throughput.
- Each CootServer can be accessed simultaneously by any number of CootHTML- Client nodes.
- Eventually, we will also provide access from CootGUIClient nodes.



Outline

- Introduction
- Choosing a System Topology
- Designing for Concurrency
- Designing for Security Policy
- Partintioning Software
- Exercise 11

Designing For Concurrency

- Concurrency means “have many things happening at once”

Designing For Concurrency

- Concurrency introduces the following issues
  - How to ensure that information is updated completely before anyone can act on the update
  - How to ensure that information is not updated while it's being read
- At a low level, database transactions and thread monitors are used to protect data inside individual processes.
- At a higher level, we need to use system rules and business rules to control concurrent activity
- If you can think of a concurrent situation that might cause difficulties for your system, do not proceed to implementation until you can guarantee that situation is no longer a problem.

Outline

- Introduction
- Choosing a System Topology
- Designing for Concurrency
- Designing for Security Policy
- Partintioning Software
- Exercise 11

Designing For Security (1)

- A secure system is on that is protected from misuse, regardless of whether the misuse is accidental or malicious
- Security can be broken down into the following five aspects:
    - Privacy: We must be able to hide information, making it available only to those who are authorized to read it (or change it)
    - Authentication: We need to know where each piece of information came from, so that we can decide whether or not to trust it
    - Irrefutability: This is the flip-side of authentication, ensuring that the originator of information can't deny that they're the source
    - Integrity: We must be sure that information hasn't been damaged, accidentally or maliciously, on its way from the source to us
    - Safety: We must be able to control access to resources (such as machines, processes, databases and files)

Designing For Security (2)

- Cryptography includes digital encryption and decryption
  - Encrypt information means to scramble it so that it's useless if anyone manages to steal it
  - The reverse of encryption is decryption. The scrambling method must be known to the intended recipient
- The following four security aspects can be implemented via using cryptography:
  - Privacy: In digital encryption and decryption, safe distribution of keys is achieved using **public/private key pairs**, **certificates** and **certificate authorities**
  - Authentication: This relies on being able to prove the origin of the key, using **certificates** and **certificate authorities**
  - Irrefutability: Once we've authenticated a piece of information, our authentication is irrefutable
  - Integrity: First, we transmit the encrypted information and the unencrypted information to the client. Next, the client decrypts the encrypted version and compares the result with the unencrypted version – obviously, the two should match. Therefore, we're confident that we have received the correct information

Designing For Security (3)

- Here are some general security rules:
  - Protect your servers from unauthorized access, whether accidental or malicious.
  - Confine sensitive information to your internal network (sensitive information includes details of business deals with other companies; business strategy; personnel details; details of the credit reference agencies you use; information relating to national security; and so on)
  - Prevent the eavesdropping of exported information (ensure that information you pass outside your intranet can only be read by the intended recipient)
  - Protect employee and customer passwords, which are not only the foundation of your entire security policy. They're often highly personal
  - Prevent server code accessing unneeded resources
  - Prevent client code accessing unneeded resources (we want to protect the client against unauthorized access to their resources and against accidental damage)

Outline

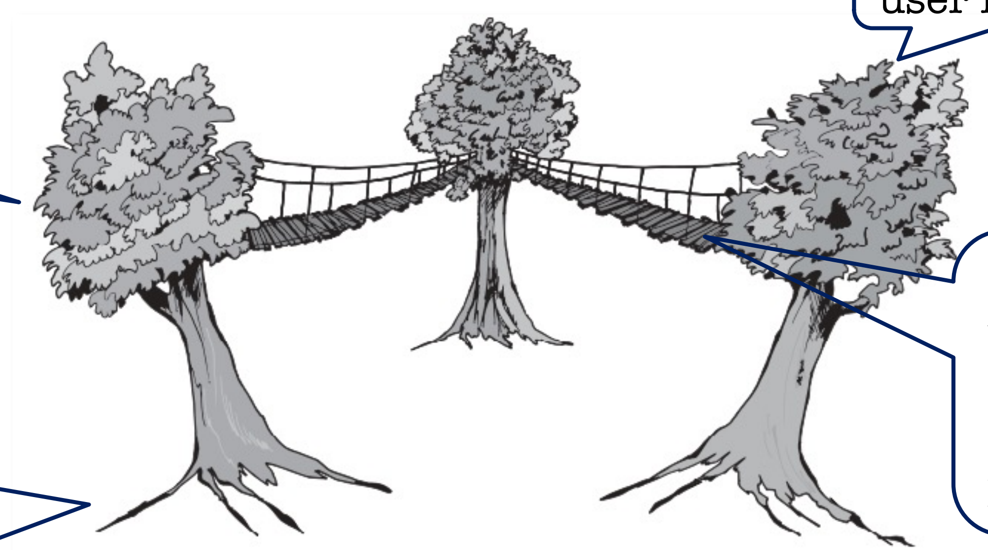
- Introduction
- Choosing a System Topology
- Designing for Concurrency
- Designing for Security Policy
- Partintioning Software
- Exercise 11

Systems and Subsystems

- Consider how the simple concept of Customer is viewed differently by the departments in a large organization, sales, marketing, billing, procurement, dispatching and so on
- If we tried to put together a single software system that supported all of these departments, our Customer would have hundreds of attributes and hundreds of operations
- Instead, a business should have a number of sperate systems, each implemented by a different development team so that the temptation to reuse objects inappropriately is minimized

The top is the user interface

A company's systems as independent trees in a forest



Communication takes place along narrow, constrained pathways

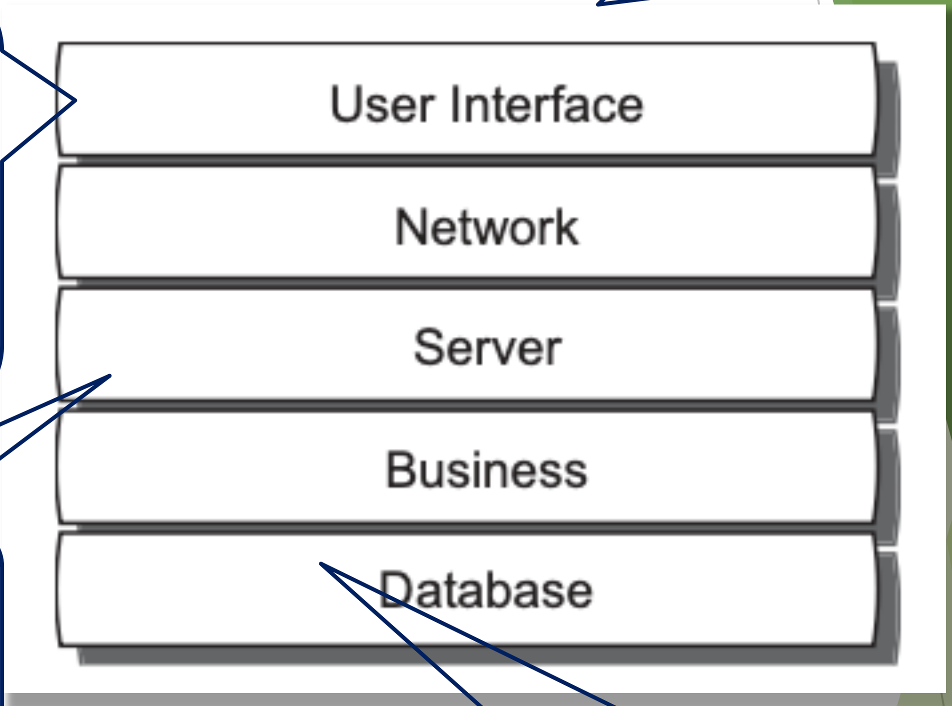
Underneath each tree is the database of information that the systems need to access

Layers (1)

- Inside a software system, we usually employ multiple layers of code
  - Each layer is a cluster of collaborating objects dependent on the facilities offers by lower layers
  - Often, regardless of the total number of layers, the top layer represents the user interface and bottom layer represents the operating systems, or a network connection

Layers (2)

Layers for Two-and Three-Tier Systems



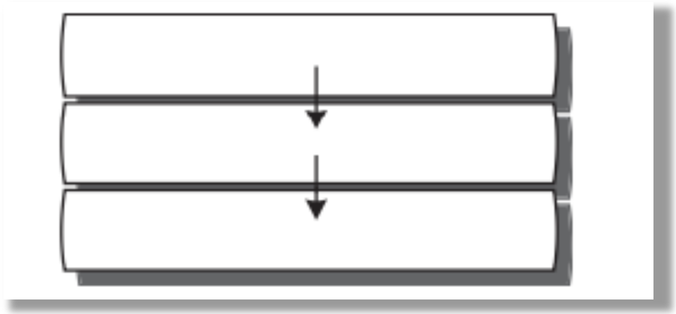
The user interface layer contains objects whose job it is to present available options to the user, to pass user commands and data on to the business layer and to display data coming back from the business layer

The business layer consists of the entity objects and supporting implementation objects

The job of the database layer is to ship data back and forth between the DBMS and business layer

Message Flow in Layers

- In a layered system, each layer is a client of the layer immediately below it
  - Messages will be sent from the upper layer to the lower layer
  - Each message is either a question or a command
- Many commands sent into a layer will have an effect on the information managed by that layer
  - What if the upper layer needs to know what information has changed?

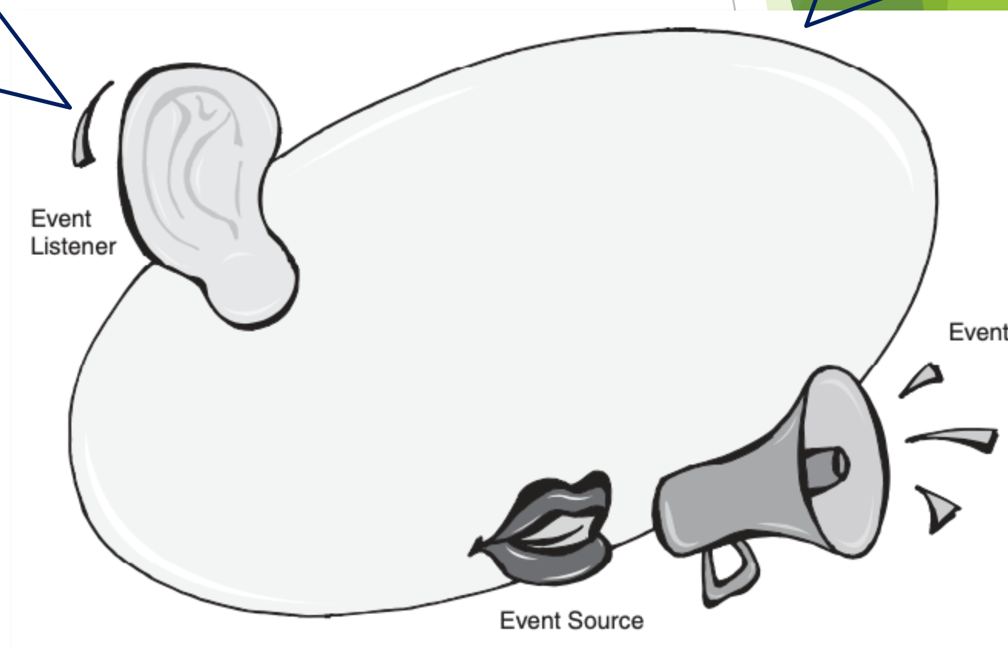


Events (1)

- Events is a way that a lower layer can use to notify the upper layer when something interesting has happened, without increasing complexity or coupling in either direction

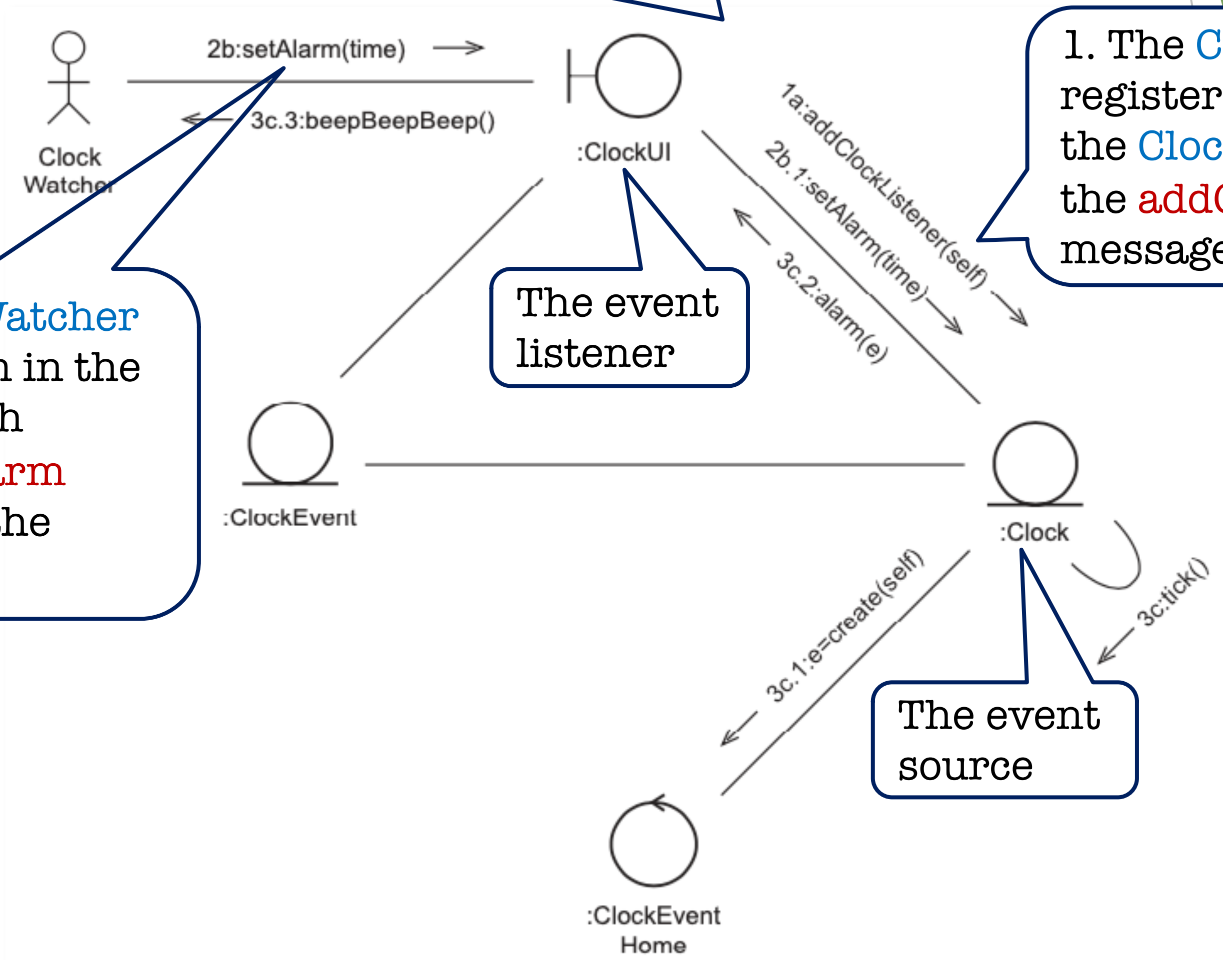
An event source detects when something interesting has happened (an event) and shouts out the details (broadcasts) to anyone who might be listening (event listeners)

A real-world analogy for events



Events (2)

An example collaboration between a Clock: a clock could broadcast an event when the time changes (every minute) and it could broadcast an event when the alarm goes off



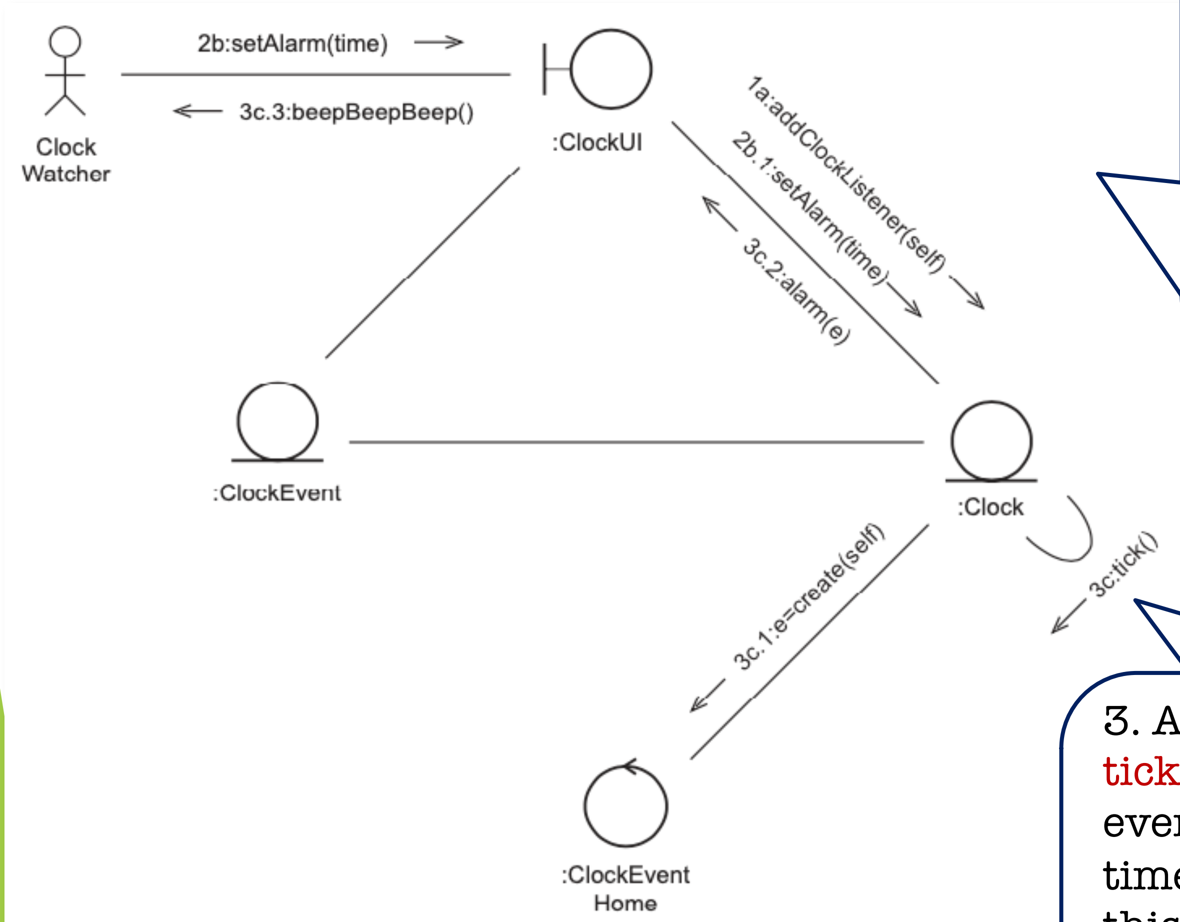
1. The ClockUI registers itself with the Clock by sending it the addClockListener message

2. The ClockWatcher sets the alarm in the ClockUI, which passes the alarm setting on to the Clock

The event listener

The event source

Events (2)



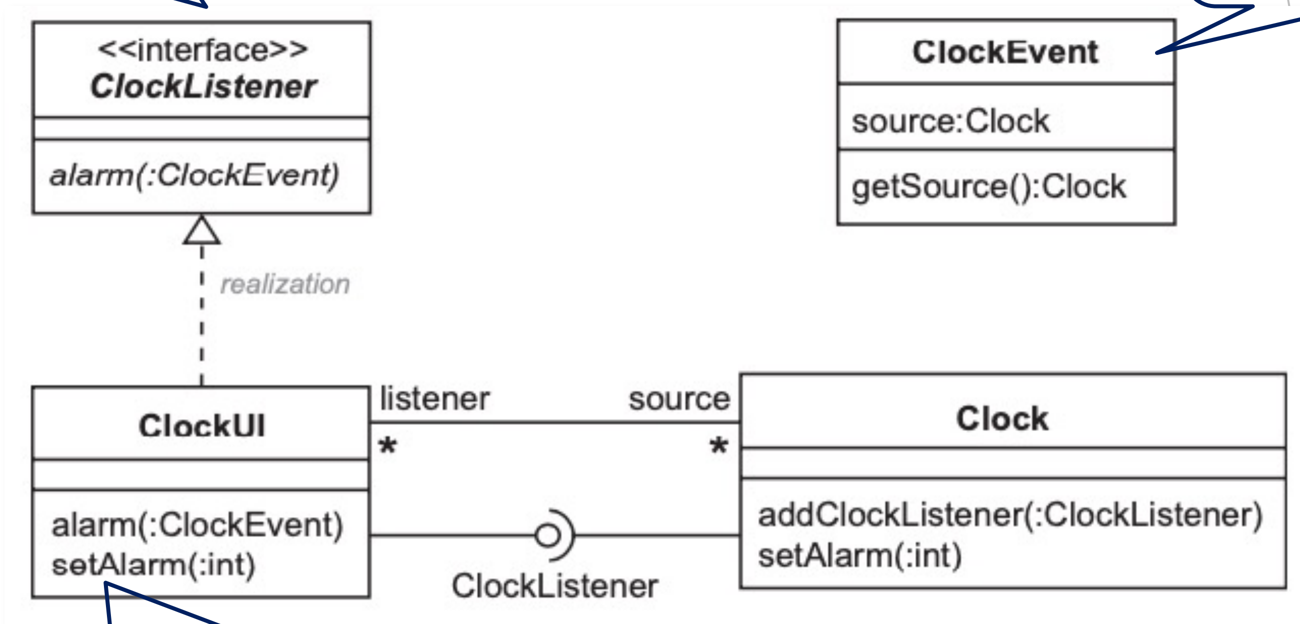
It can show the ordering of independent messages by numbers: (for example, it is implicit that message *2b* happens before message *3c*; it would also be implicit that messages numbered *99x* and *99y* happen **at the same time**)

3. As the clock sends itself the tick message periodically, eventually it will detect that it is time for the alarm to go off. When this happens, the Clock creates a ClockEvent (with the help of the ClockEventHome), with information about the event

Events (3)

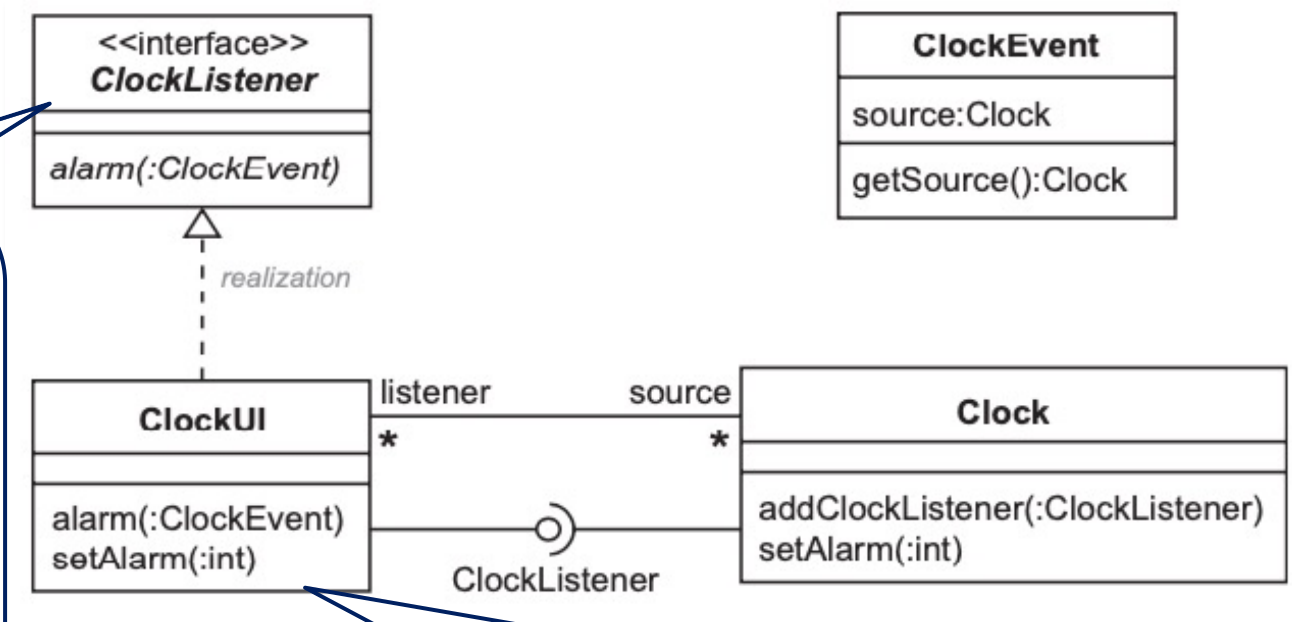
The class diagram for the alarm clock scenario

ClockEvent has a getter for the source attribute



The ClockUI has a message that allows the ClockWatcher to set the alarm (setAlarm) and another message for detecting the alarm event (alarm). Finally, the Clock class has a message for setting the alarm (setAlarm) and a message for registering a listener (addClockListener)

Events (3)



An **interface**, denoted by the *<<interface>>* keyword, is a pure abstract class. The dashed arrow with a white head indicates inheritance, for the special case where the superclass is an interface.

Because OO languages generally don't have a true broadcast mechanism, the listeners, for their part, must make sure that they register for the event. ClockListener is an abstract class, that only lists the messages required for detecting Clock events. As long as ClockUI inherits from ClockListener, we will be able to register a ClockUI with a Clock and the ClockUI will be able to receive the alarm message. Thus, although Clock *is* coupled to ClockListener, it is *not* coupled to ClockUI.

Outline

- Introduction
- Choosing a System Topology
- Designing for Concurrency
- Designing for Security Policy
- Partintioning Software
- Exercise 11

Exercise 11

- Deadline: **2022/06/30 (Thur.) 9:00**
- Please submit your answer file "UML_Ex11_Your group names.pdf" to "Exercise 11" under "Assignments" tab in Manaba +R
- The maximum points for "Exercise 11" will be **5p**
- If you put a wrong file name or wrong file format, your assignment will not be evaluated. Please be careful!

Ex11 (Group work)

- In the last two week, each group selected their own business and did the requirement analysis and the problem analysis for the business.
- This week, it is a searching task:
  1. Please search what a client, a server and database is
  2. Please search popular programming languages for building a client, a server and database
- For the searching task, there are following requirements
  1. Please report your own understanding about a client, a server and database
  2. Please report at least three programming languages for each.
  3. Please compare the programming languages with each other for each group in some sides, such as, history, benefits and disadvantages, etc.
  4. The report in total should be at least 400 words.