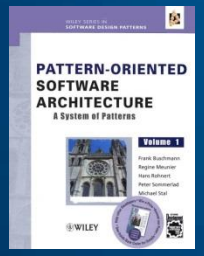
Digital Transformation: CSC4008

Software Architecture

Learning Outcomes

Appreciate the factors in making a choice of software architecture (tactics)

Know a range of Software Architecture Styles and Patterns



Ref: Larman Chapter 29
Software Architecture: Foundations, Theory, and Practice – Taylor et al. Ch.6
L.Bass et al - Software Architecture in Practice

Definition of Software Architecture

- “Software Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.” [ANSI/IEEE Std 1471-2000]

- “Architecture represents the set of significant design decisions that shape a system, where significant is measured by cost of change” [Booch, Patterns, Patterns and More Patterns]

- The Software Architecture of a system is the set of *structures* needed to **reason about the system**, which comprises software elements, relations among them and properties of both [SEI – Carnegie Mellon University]

Why?

What

Requirements

Detailed Design

- Cf performance requirement of 100 concurrent transactions per second
- Helps decide on architecture – need transaction processing component separate and to avoid contention on storage/other resources
- Also – reuseability, flexibility, security, portability … can be considered

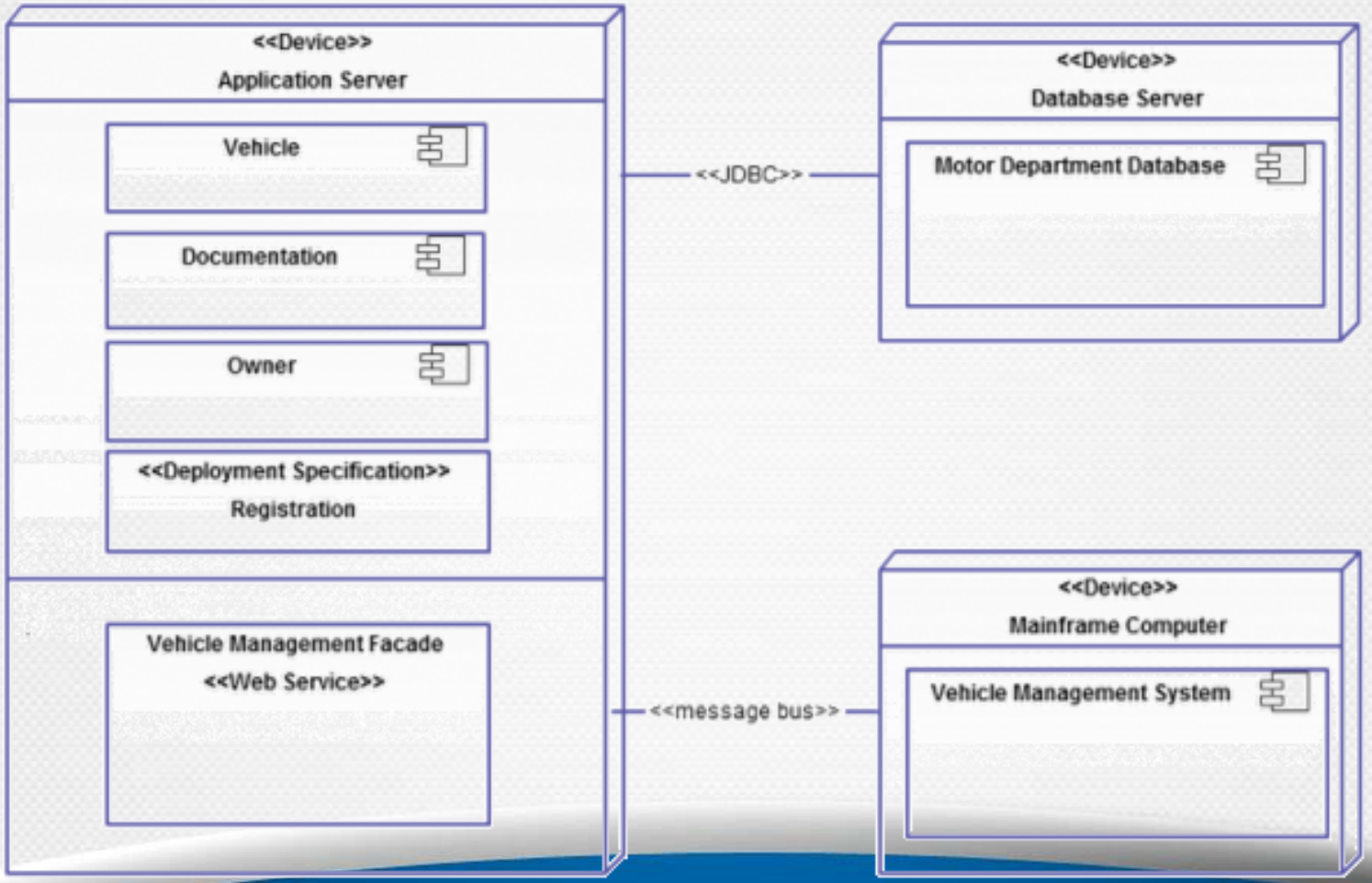
Modelling Architecture

- Natural Language
  - Expressive BUT Not machine readable
- Example – Lunar Lander Simulator
  - The Lunar Lander application consists of 3 components: a data store component; a calculation component; and a user interface component.
  - The data store component will store and allow other components access to the height velocity and fuel of the Lander as well as the current simulator time.
  - The calculation component, upon receipt of a burn rate quantity retrieves current values of height, velocity and fuel from the data store component, updates them with respect to the input burn rate, and stores the new values back. It also retrieves, increments and stores back the simulator time. It is also responsible for notifying the calling component of whether or not the simulator has terminated and with what state (landed safely, crashed, etc)
  - The UI component displays the current status of the Lander using the information from the data store and calculation. While the simulator is running, it retrieves the burn rate from the user and invokes the calculation.

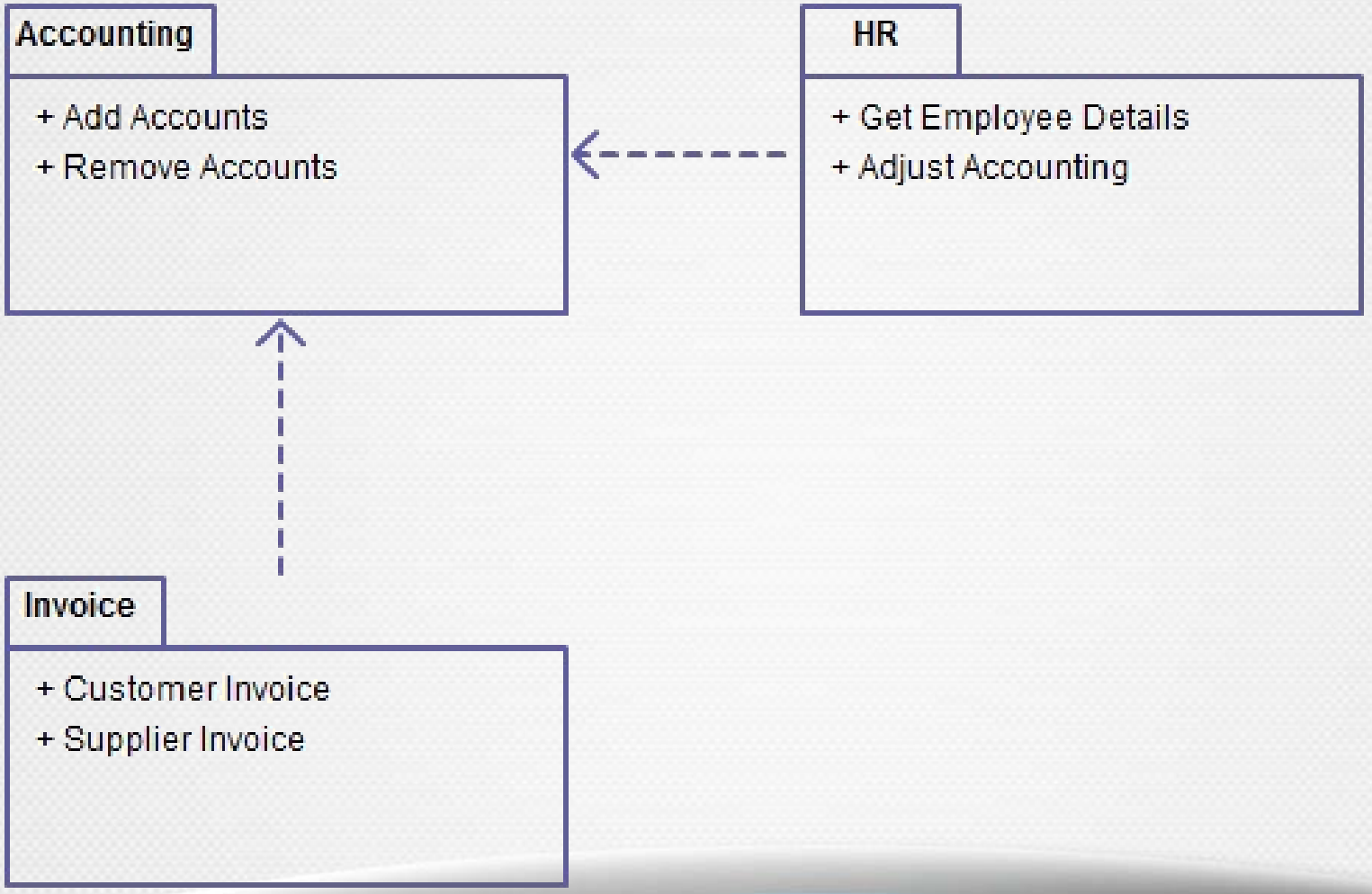
Software Architect Document - N+1 (or 4+1) View model

- 4+1
  - Logical – most important layers, subsystems, packages, classes etc
    - Focuses on functionality ie. what the system must do to satisfy stakeholders
    - class diagrams are useful
  - Process – main processes
    - Focuses on non-functional requirements ie. quality attributes such as performance, reliability etc.
    - interaction diagrams (sequence diagrams/collaboration diagrams)
  - Development
    - describes the actual software structure e.g. programming language, libraries, frameworks etc
    - details of software development, project management
  - Physical
    - UML deployment diagram showing mapping to physical architecture
  - +1 = scenarios/use case view
    - most important scenarios for the architecture
    - Text using the components in the other views – enough to explain the architecture (not full requirements spec!)
- N+1 = above plus other views
  - E.g. Security – http authentication, database access etc

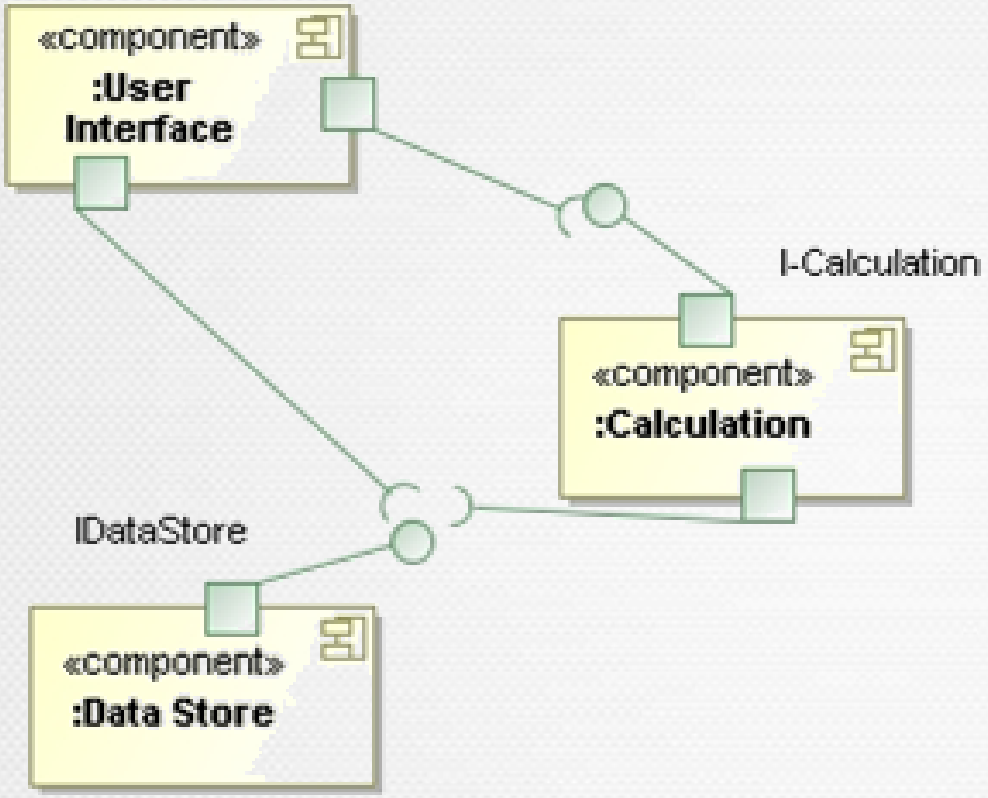
UML Deployment Diagram



UML Package Diagram



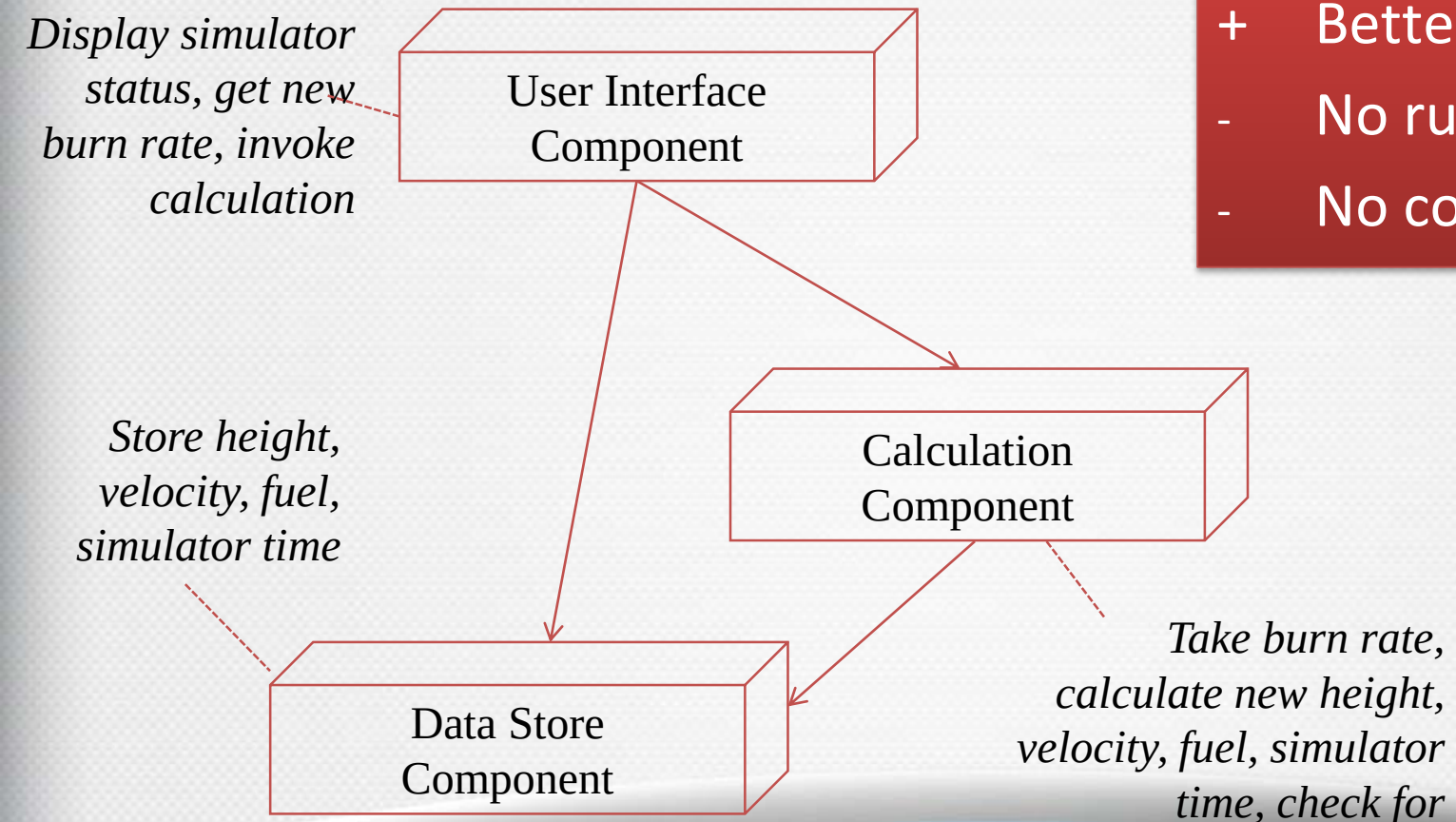
UML Component Diagram



- Shows same but with interface (connectors) defined better
- Ball = provided interface
- Socket = required interface

Modelling Architecture

- Informal Diagram



+ Easy/quick
+ Better than NL
- No rules/ semantics
- No consistency checking

Architecture Document

1. Architectural Goals – functionality/problem statement

2. Architectural Significant Requirements
   2.1 Functional
   2.2 Non-functional
3. Decisions and Justification - explanation
4. Key Abstractions/Domain Model - big building blocks + major abstractions used + important
   trade-offs + critical subsystems

5. Software Partitioning i.e. 4+1 model
   see earlier

Architectual Style or Pattern?

- Usually people mean the same by these 2 terms
- But often a *style* means a way of organising (partitioning) your code, while a pattern is a known solution to a known problem

Higher abstraction

- Styles Examples
  - Monolithic
  - Layered
  - Pipes and filters
  - Event-driven
  - Publish-subscribe
  - Client-server
  - SOA
- Patterns Examples
  - N-tier
  - Microkernel
  - Model-View-Controller
  - Model-View-ViewMode

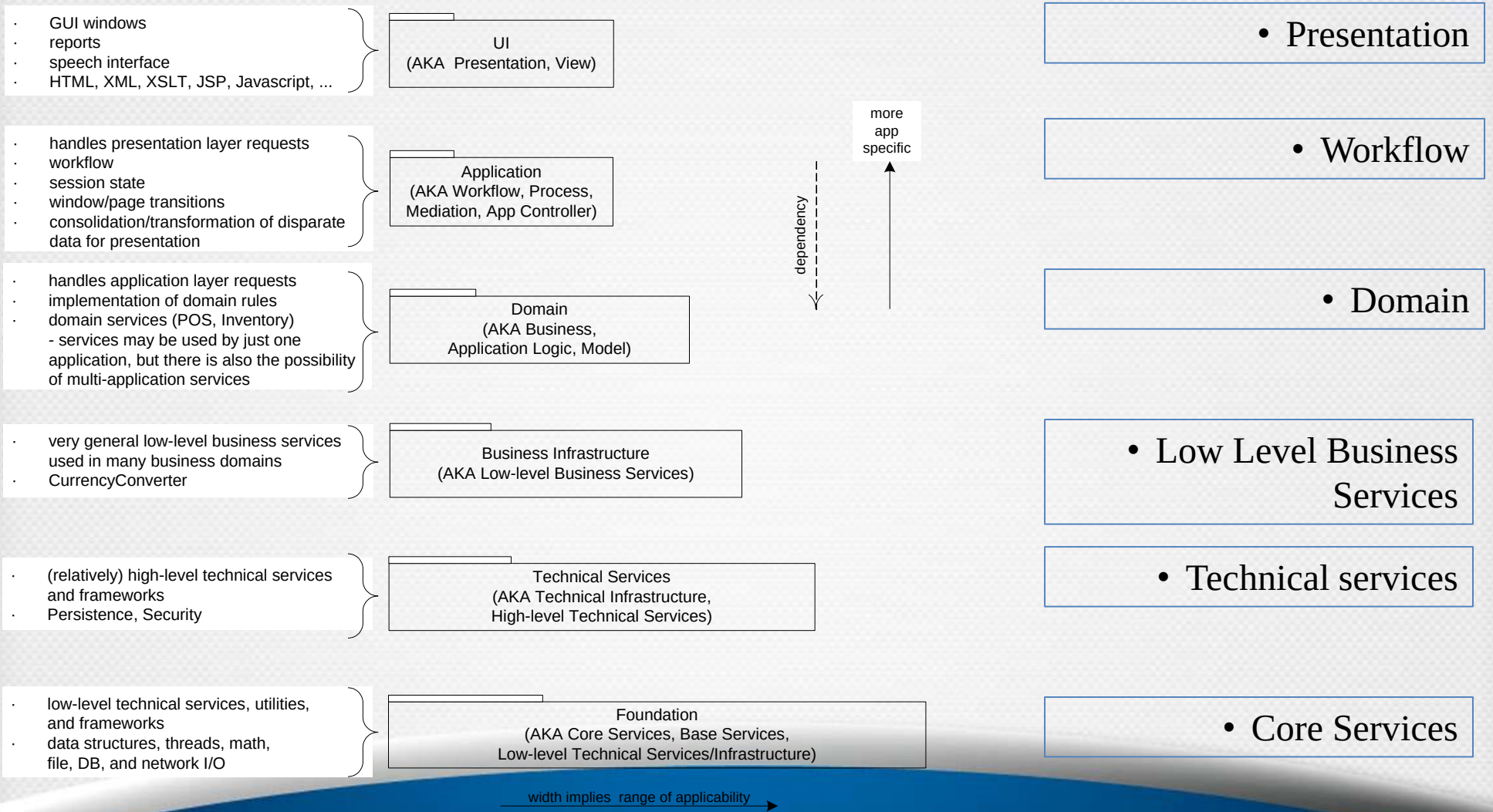
Layers Style

- Layers Style
  - Organise the large-scale logical structure of a system into discrete layers of distinct, related responsibilities with a clean cohesive separation of concerns
  - Normally high layers are application specific, low layers are low level services
    - Coupling is from high level → low level
    - Avoid low level → high level coupling
  - Why?
    - Keeps coupling low = more reuse, easier maintenance
      - Source code changes easier
      - Keeps UI separate from application logic so that application logic reuseable
      - Services can be grouped in a layer and reused

[1] F. Buschmann et al., *Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons*,1996

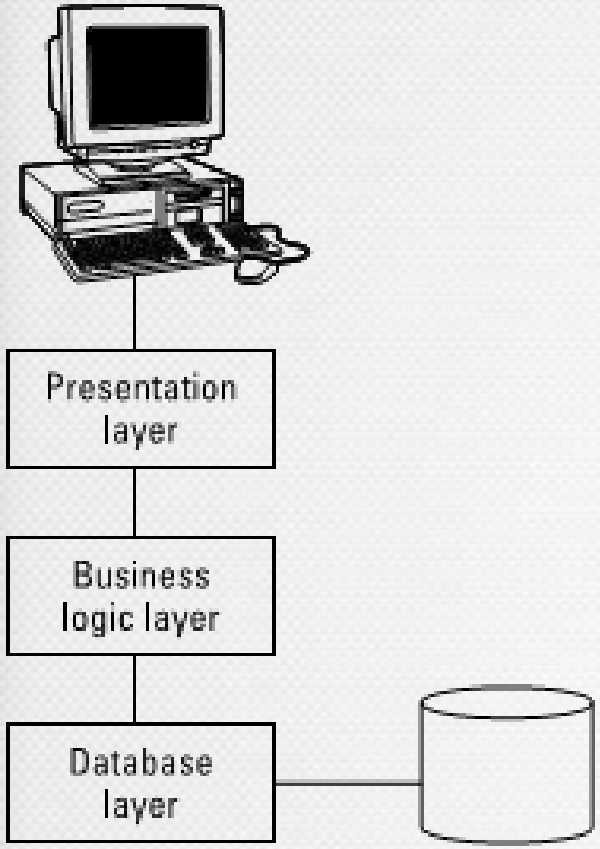
Layers

Example:



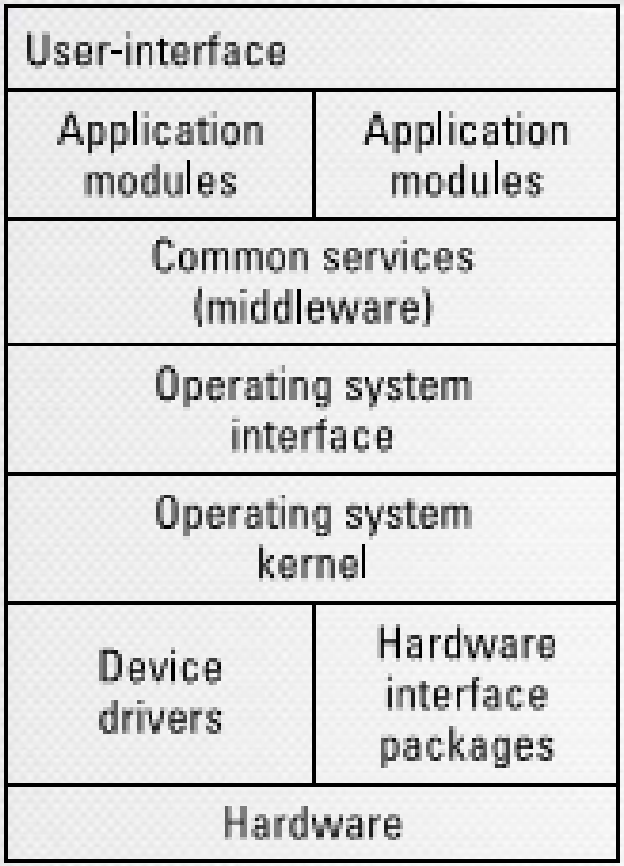
- Presentation
- Workflow
- Domain
- Low Level Business Services
- Technical services
- Core Services

Web Application using Layers



- **Presentation** layer = web server that delivers content to the user's browser
- **Business logic** layer – e.g. is built on JBoss application server platform.
  - receives requests from the presentation layer, processes the requests, and supplies the results to the presentation layer
  - requests the data that it needs from the database layer
- **Database** layer manages persistent data e.g. customer data, inventory etc
  - supplies this data to the business logic upon request.

Operating Systems as Layers



- Add a new device?
  - e,.g. storage
- Just add a new driver – kernel, operating system interface … user interface ‘don’t care’

Benefits of Layer Architecture

- Separation of concerns – reduces coupling, promotes cohesion, aids reuse, easier maintenance, easier understanding
  - E.g. UI objects should not do application logic (Model-View separation)
  - principle of least knowledge
- Different layers can have different authorization (**sandboxing**)
- Related complexity encapsulated
- Some layers (typically higher ones) replaceable e.g. new business rules in the business domain
- Some layers can be distributed e.g. domain and technical services
- Division of work – teams
- Maps to implementation – packages in Java, Namespaces in C++/.Net

Disadvantages of Layering

- **Layers aren't as efficient as hard-coded connections inside a monolithic solution. Direct calls are quicker than via layers**
- **Protocol stack layers increase message size and add processing time – especially distributed layers**
  - Sending a message may involve headers ∴ bigger message size/ slower transfer of information
- **Changes in a layer sometimes cascade into other layers.** E.g. where one layer becomes faster
- **Layers sometimes introduce unnecessary work.** e.g. When the same request is sent several times. E.g.suppose a layer sends a checksum each time

Creating the Software Architecture

- Choosing an architecture tightly linked with Non-functional requirements
  - Technical constraints e.g. Must be in C#, run on Windows 7, etc
  - Business constraints e.g. Must interface to SAP
  - Quality attributes:
    - scalability,
    - availability,
    - ease of change (modifiability)
    - portability,
    - usability,
    - performance ...

Creating the Software Architecture

- Choosing an architecture tightly linked with Non-functional requirements
  - Technical constraints e.g. Must be in C#, run on Windows 7, etc
  - Business constraints e.g. Must interface to SAP
  - Quality attributes:
    - scalability,
    - availability,
    - interoperability
    - ease of change (modifiability)
    - portability,
    - usability,
    - Performance
    - security ...

Quality attributes – Performance

- Throughput
  - Work done per time unit e.g. Transactions per second (tps) or messages per second (mps)
- Response Time
  - Latency = time to respond to some event e.g. an input
  - *Guaranteed response time* means all requests
  - *Average Response time* also common
    - E.g. 95% of requests processed in <4s and none >15s
- Deadline
  - Where a process has a window of time to finish in
- Memory Usage

Quality attributes – Performance tactics

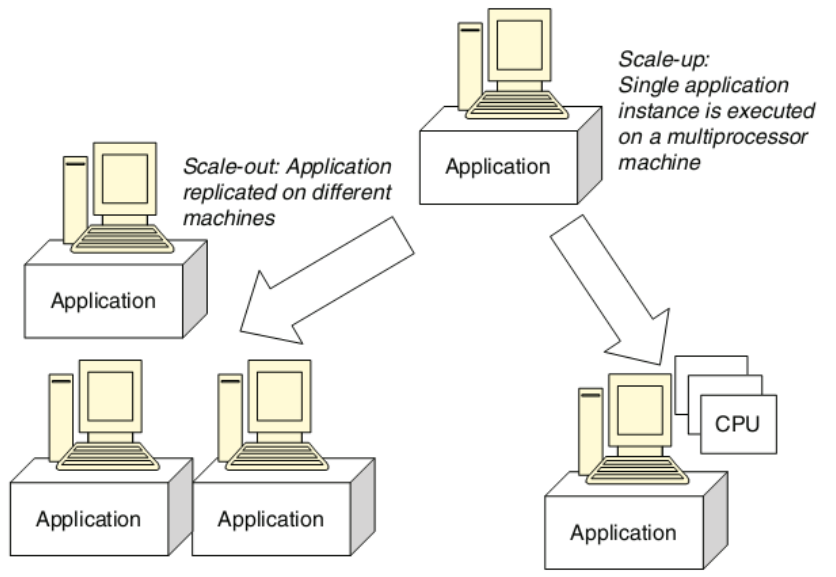
- 2 sources of performance hit
  - Resource consumption.
    - CPU, data stores, network bandwidth, and memory, runtime buffers ... All limit performance i.e. Introduce latency
  - Blocked time.
    - Due to **contention** for resources.
    - Due to **non or limited availability** of resources (e.g. due to network or maintenance issues)
    - Due to **dependency** on other computation. I,e, having to wait for results elsewhere

Quality attributes – Performance tactics

- Increase computational efficiency.
  - E.g. Faster algorithms in critical areas, intermediate data in repository, ...
  - Reduce computational overhead. E.g. using classes rather than (RMI),
  - Don't use intermediaries (nb tradeoff with modifiability Manage event rate e.g. sampling frequency for monitoring
- Bound execution times - limit on how much execution time is used to respond to an event. e.g. Optimisation algorithm
- Bound queue sizes – controls maximum number of queued arrivals - implies rejected requests
- Concurrency.
  - load balancing
- Caching
- Resource Arbitration - FIFO, prioritisation of requests

Quality attributes - Scalability

- How well a solution to some problem will work when the size of the problem increases



» Scale Up = s/w architecture could allow more hardware to be added (more (or faster) CPUs, more memory, more storage ...)

- Works well on multi threaded/ multiple single threaded process instances

» Scale out – s/w architecture is such that all or part of the system can be distributed to other machines

Quality attributes – Scalability cont.

- Simultaneous Connections
  - E.g. architecture designed to support 1,000 concurrent users
  - Need care to ensure not all using the same resource (e.g. Disk)
- Data Size
  - What happens if a file is encountered bigger than normal
  - Database grows bigger
- Deployment
  - s/w architecture has to be deployed
  - Must be easy to configure if many users / versions

Quality attributes – Modifiability

- How easy it may be to change an application to cater for new functional and non-functional requirements
  - Need estimate of effort for change
  - Loose coupling makes change easier/ cheaper
  - But making coupling loose
    - Especially - partition to keep volatile functionality decoupled
  - Localising modification – separate the part that varies
  - High Cohesion/Low Coupling

Quality attributes – Security

- Authentication : verifying identity of users and other applications with which they communicate.
- Authorisation : defined access rights – users & applications
- Encryption
- Integrity - contents of a messages in are not altered in transit.
- Non repudiation : The sender of a message has proof of delivery and the receiver is assured of the sender's identity.
  - This means neither can subsequently refute the message exchange

Quality attributes – Security tactics

- Resisting Attacks
  - Authenticate users - passwords, digital certificates, biometric id ...
  - Authorize users - ensuring that an authenticated user has the rights to access data or services.
    - Access control by user /user class, user groups, user roles, individuals
- Maintain data confidentiality.
  - encryption to data and to communication links
- Maintain integrity - Data should be delivered as intended
  - Checksums, hash results, encrypted
- Limit exposure - limited services on each host
- Limit access - Firewalls restrict
- Detecting attacks – usually via an intrusion detection system.
  - comparing network traffic patterns to a historic patterns

Quality attributes – Availability

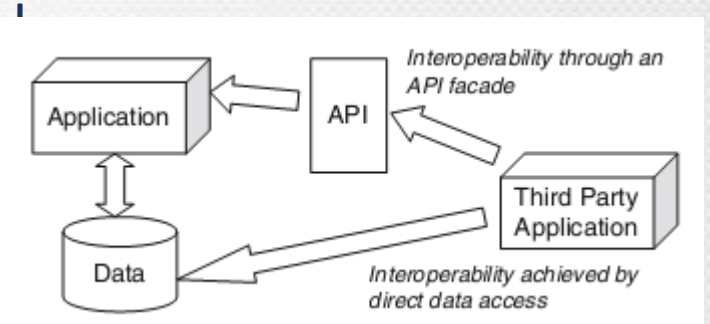
- Related to reliability
  - Failures lead to unavailability – availability measured in uptime/total time. Also Mean time to Failure (MTTF)
- Recoverability also important - Time to detect and repair
- Fault Detection
  - Ping/echo - Issue a ping, expects to receive back echo in time t
  - Heartbeat (dead man timer) - emit a heartbeat message periodically and another component listens for it. If the heartbeat fails, the originating component is assumed to have failed and a fault correction component is notified. (heartbeat could also carry data e.g. log of the last transaction)
- Fault Recovery
  - Redundancy
    - Typically for control systems
  - Checkpoint/rollback

Quality attributes – Availability + Tactics

- **Fault Prevention**
  - Removal from servic e.g. periodic rebooting to prevent memory leaks building up
    - architectural strategy can be designed to automate the removal.
  - Process monitor – fault detected delete the nonperforming process and create a new instance of it

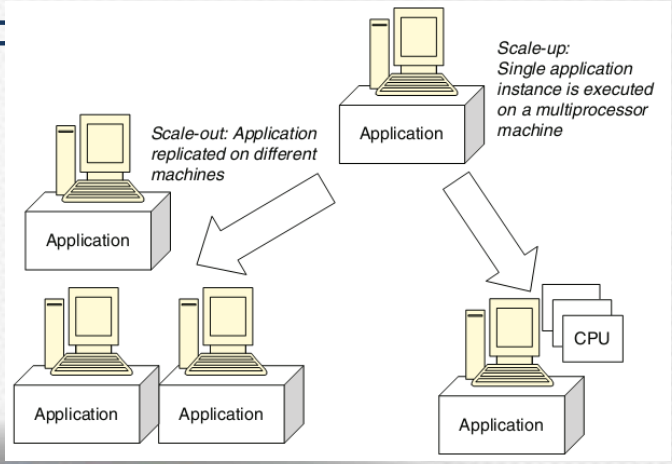
Quality Attributes - others

- Integration – how it fits with other (external or legacy) systems
  - Commonly an API is used
  - Or data integration – e.g. Via rDBMS or XML
  - API is better from point of view of contro
- Portability
- Testability : Architecture decisions can greatly affect the amount of test case s that are required.
- Supportability : Support typically involves diagnosing and fixing ∴ loose coupling and good error handling (logs) help



Example Quality Tactics

- Things we do to achieve certain values of attributes
  - Performance attributes: Throughput, Response Time (e.g. latency)
  - Performance Tactics: Increase computational efficiency; Concurrency, Caching, Bound queue sizes (ie reject requests)
  - Scalability Attributes: simukltateous connections; data size, software architecture choice
  - Scalability Tactics: Scale-up (Memory↑, CP
    via distributed
  - Security Tactics: Authentication, Authorisation, Encryption, Non-repudiation



Sample Architecture quality requirements

Quality attribute	Architecture requirement
Performance	Application performance must provide sub-four second response times for 90% of requests
Security	All communications must be authenticated and encrypted using certificates
Resource management	The server component must run on a low end office-based server with 2GB memory
Usability	The user interface component must run in an Internet browser to support remote users
Availability	The system must run 24 × 7 × 365, with overall availability of 0.99
Reliability	No message loss is allowed, and all message delivery outcomes must be known with 30 s
Scalability	The application must be able to handle a peak load of 500 concurrent users during the enrollment period
Modifiability	The architecture must support a phased migration from the current Forth Generation Language (4GL) version to a .NET systems technology solution

Architectural Patterns

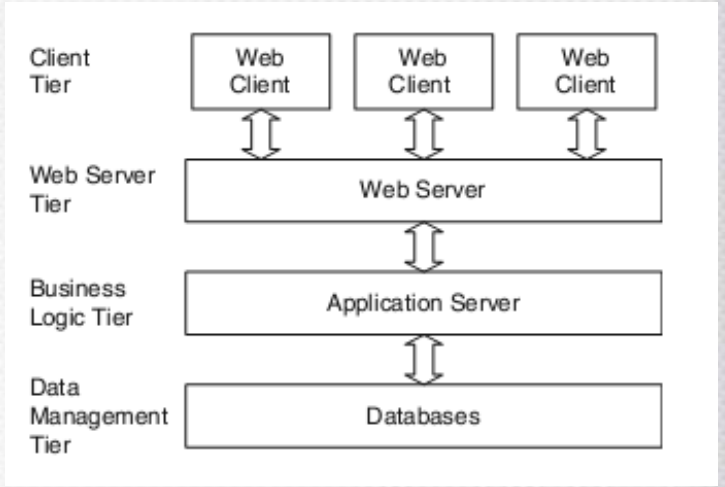
- Architectural Pattern has:
  - A set of element types (such as a data repository or a components).
  - A topological layout of the elements indicating their interrelationships.
  - A set of semantic constraints (e.g., filters in a pipe-and-filter style are pure data transducers—they incrementally transform their input stream into an output stream, but do not control either upstream or downstream elements).
  - A set of interaction mechanisms (e.g., subroutine call, event-subscriber, blackboard) that determine how the elements coordinate through the allowed topology

Case Study – easybacklog/trello/…

- Characterise the architecture style
- Characterise the interactions – how do we get the server side to do the things we want – how does it get the message?
- How does it keep details of me?
- How do we get all this data back?

N-tier (multitier) Architectural Pattern

- Layer style - “client/server” = relationship between adjacent tiers
- Key features/tactics
  - Commonly use “request-response”
    - synchronous or asynchronous
  - 2-tier – clients – server
- 3-tier usually client-business/application-storage
- Key features/tactics
  - Separation of concerns: Presentation, business and data handling logic partitioned
  - Flexible deployment : tiers can run on the same machine, or each on its own machine
- Disadvantages
  - comms overhead
  - single point of failure



N-tier quality attributes

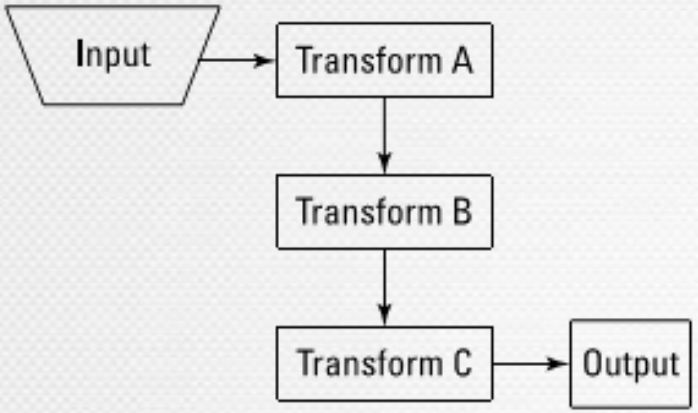
- Availability – each tier can be replicated. In case of failure application keeps running
- Failure Handling – Web & application servers often have *transparent failover* meaning redirection to replica server
- Modifiability – Separation of concerns enhances this. Change to any layer should not affect other layers (see notes on Layers)
- Performance – well established architecture – good performance.
  - Issues: Number of concurrent threads on each server, speed of connection between tiers; amount of data
- Scalability – easy – can have multiple server instances & can replicate servers as required.
  - Issue often with data layer (hard to replicate, so becomes a bottleneck)

N-tier Advantages

- Availability – each tier can be replicated. In case of failure application keeps running
- Failure Handling – Web & application servers often have *transparent failover* meaning redirection to replica server
- Modifiability – Separation of concerns enhances this. Change to any layer should not affect other layers (see notes on Layers)
- Performance – well established architecture – good performance.
  - Issues: Number of concurrent threads on each server, speed of connection between tiers; amount of data
- Scalability – easy – can have multiple server instances & can replicate servers as required.
  - Issue often with data layer (hard to replicate, so becomes a bottleneck)

Pipe and Filter

- for applications that stream data
- E.g. Analysing an Image Stream



- Could be solved with a sequential program
- Or pipes and filters – each step cohesive, so better design + can reuse filters
- makes development easier, too. Some focus on Filter A others on Filter C

Definition of Pipe and Filter

- Uses a Dataflow architectural style
- Structures the processing of a stream of data.
  - Each processing step is implemented as a filter, with information flowing between the filters through a pipe.
  - Filters can be combined in many ways to provide a family of solutions.

Benefits of Pipe and Filter Architecture

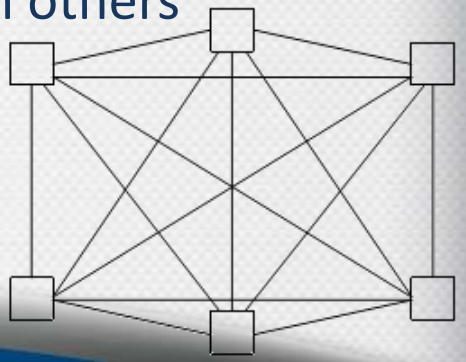
- **Flexibility -** filters can be changed / new filters added/ reordered rearranging the filters in a different order – to change the system's behaviour.
- **Reuse** -because the inputs and outputs are well defined and standard, you can use the filters created for one application for different applications and combine them with different filters
  - Facilitates rapid prototyping
- Easy to debug – use files to replace certain filters / also to speed up.
- **Performance** - Steaming approach avoids (slower) use of files where components are used instead of filters
- **Parallel processing -** filters can be started in parallel and run independently, allowing the work to be done in parallel (assuming small portions of work)

Disadvantages of Pipe and Filter

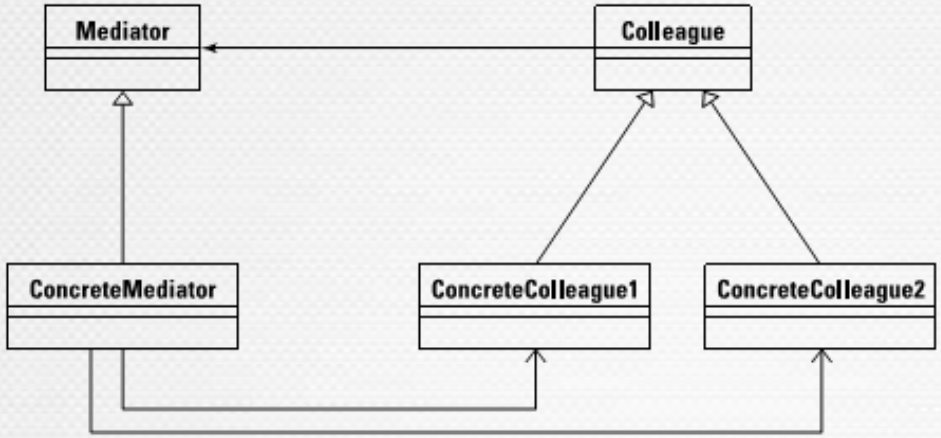
- Metadata is difficult – where information on state must also be passed this is hard to sync with the data
- Parallel processing benefits don't always materialise. Benefit is only if each filter can get going on some small part of the data. Hard to sync if dependencies.
- Error handling difficult – errors end to collapse the whole system.

Broker Pattern

- Problem scenario
  - Build a system which is a collection of component subsystems that work together to provides your company's services to its customers.
  - Components don't know about one another or even how to call each other!
  - Multiple operating systems and hardware platforms.
  - All the components are independent but must they cooperate to provide the overall service.
  - Components can act as clients to each other and use each others services
    - Cf: Everyone has a phone but no one knows other numbers
  - Could hard code each connection



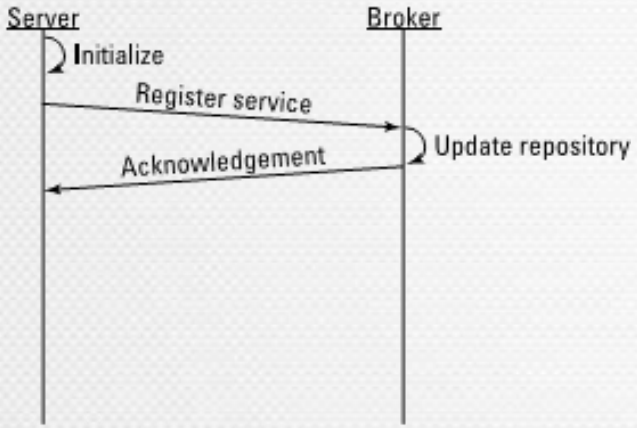
- Mediator springs to mind!



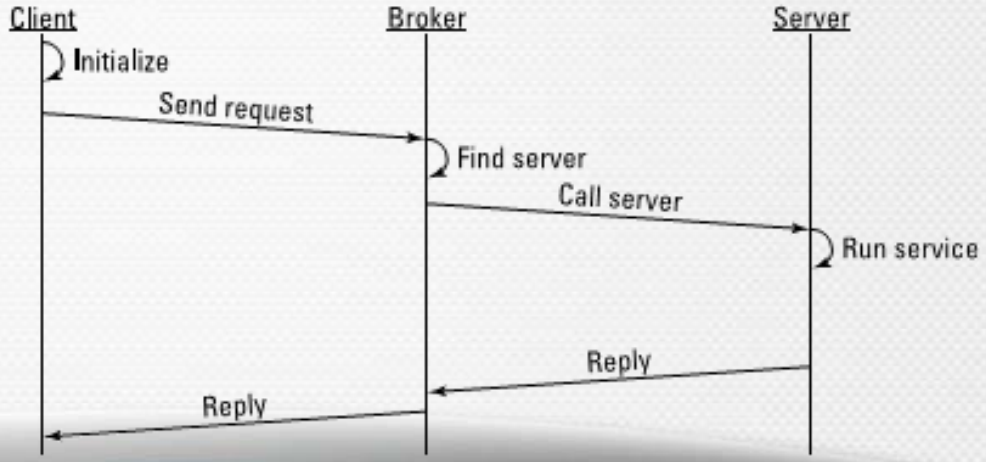
- But these are components not classes
- => Broker – like mediator but has a lookup function and is for components that offer services

Broker

- Different Clients and Servers don't 'know' each other
- A client knows some server can provide the service it needs
- Asks a middleman – the Broker
- Servers must register with the Broker



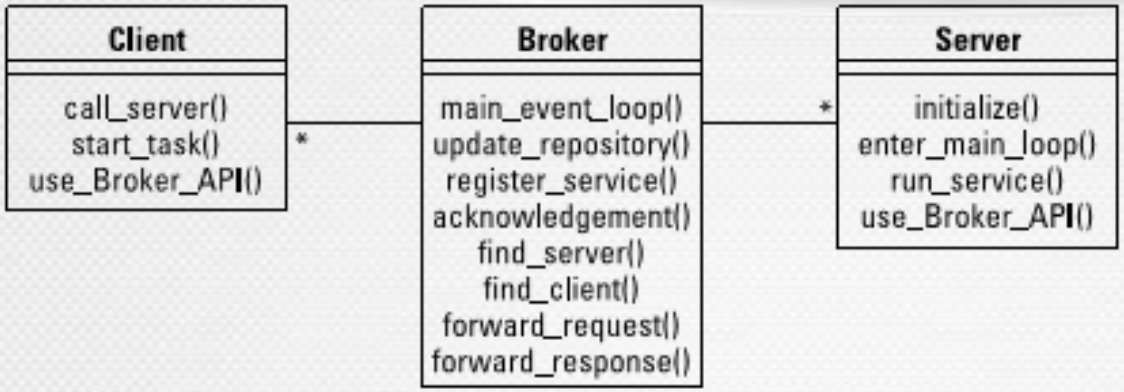
- Thereafter, a client can 'ask' for a service



Broker Variations

- Direct Communication – Broker sets up the communications and the client and server communicate directly
  - More efficient since messages do not go through the broker
- Trader – client asks for a service/ broker finds one
- Adapter – broker provides an adaptor but service provided by any one of several providers
- Callback – for reactive systems. Event arrives at broker, broker carries out service for registered clients (Observer)
- Messaging – messages come in and Broker redirects

Broker Implementation



Class Broker	*Collaborators*
Responsibilities • Register and unregister servers • Provide APIs • Transfer messages • Error recovery • Interoperate with other broker systems through bridges • Locate servers	• Client • Server • Client-side proxy • Server-side proxy • Bridge

Class Client	*Collaborators*
Responsibilities • Implement user functionality • Send request to servers through client-side proxies	• Client-side proxy • Broker

Class Server	*Collaborators*
Responsibilities • Implement services • Register itself with local broker • Send responses and exceptions to client through server-side proxy	• Server-side proxy • Broker

Roles

- Broker
  - keeps a registry of servers and their addresses
  - Allows registration/deregistration via an API
  - Routes service requests it to the appropriate server.
- Server
  - Provides a service to many clients
- Client
  - When it needs a service, the client puts its request into a message to the broker.
  - Clients need to know what server or service they want but not its location
- Note that servers may play the role of server or client and vice versa

Broker – Proxies and Bridges

- Sometimes proxy components are needed between the client-broker/ broker-server.
  - To hide interprocess communication mechanism in use
- Proxies
  - Handling communication between clients or servers and the broker
  - Translate the object model of the client (or server) into the object model expected by the broker
  - Marshalling (e.g. serialisation) parameters to go into requests and unmarshaling data from replies

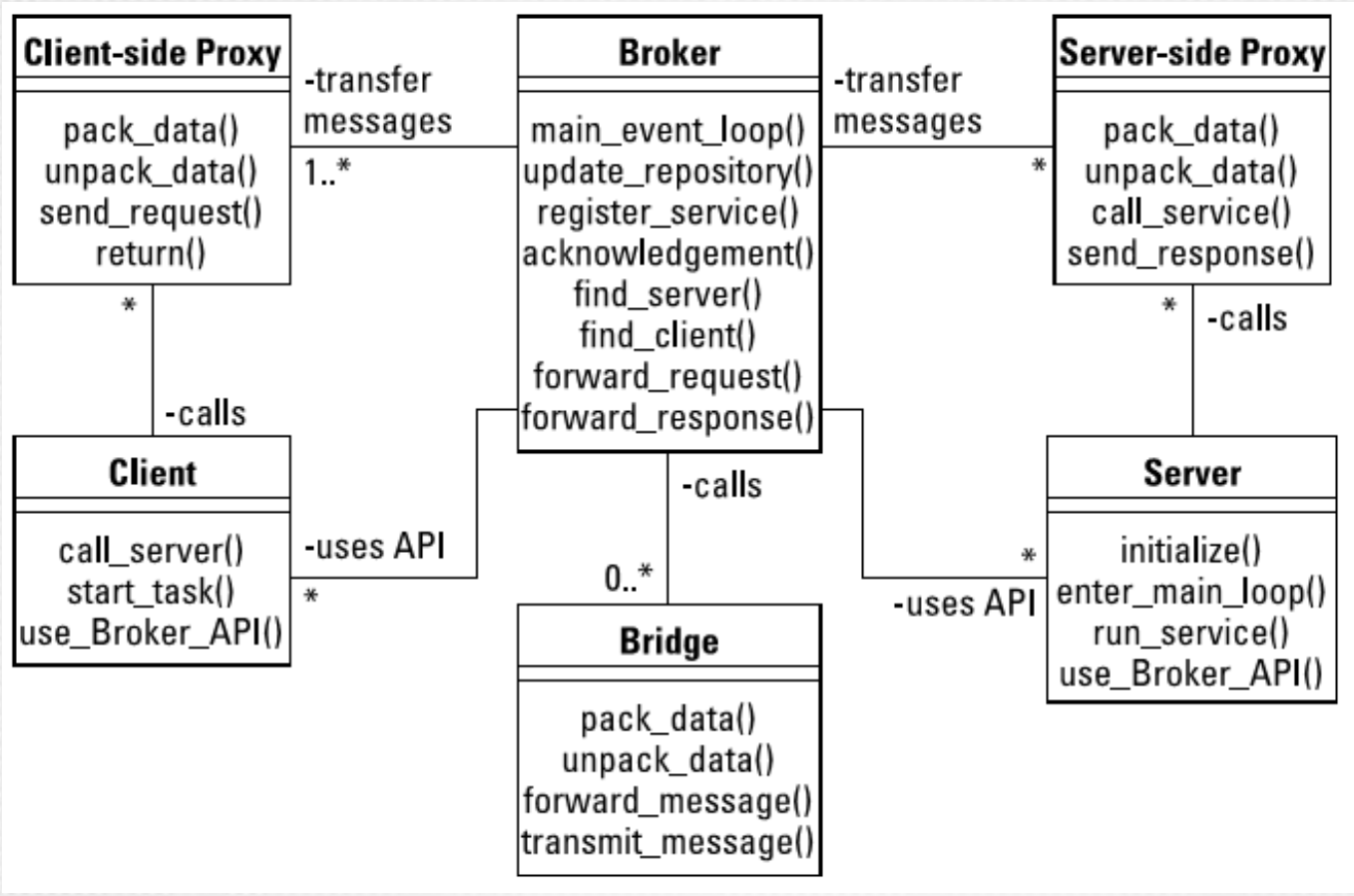
Class Client-side proxy	*Collaborators* • Client • Broker
Responsibilities • Encapsulate system-specific functionality • Mediate between client and broker	

Class Server-side proxy	*Collaborators* • Server • Broker
Responsibilities • Call services within the server • Encapsulate system-specific functionality • Mediate between the server and the broker	

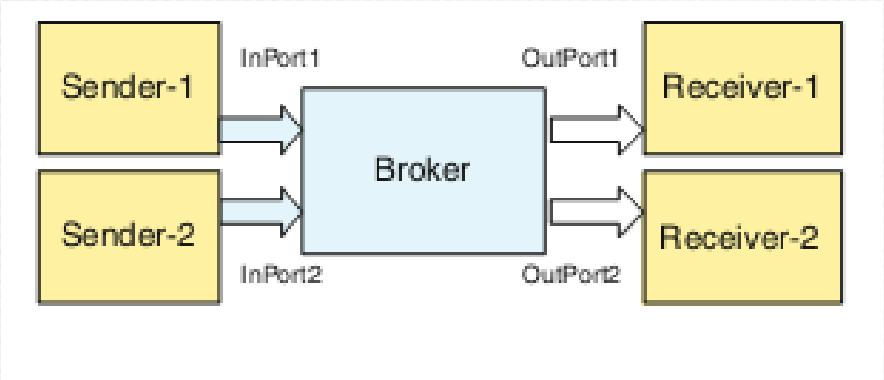
Broker - Bridges

- Sometimes a broker doesn't have a service register – it can use a different broker system
- Use a *bridge*

Class Bridge	***Collaborators*** • Broker • Bridge
Responsibilities • Encapsulate network-specific functionality • Mediate between the local broker and the bridge of a remote broker	


Client-side Proxy
pack_data()
unpack_data()
send_request()
return()
-transfer messages
1..*
Broker
main_event_loop()
update_repository()
register_service()
acknowledgement()
find_server()
find_client()
forward_request()
forward_response()
-transfer messages
*
Server-side Proxy
pack_data()
unpack_data()
call_service()
send_response()
*
-calls
-calls
Client
call_server()
start_task()
use_Broker_API()
-uses API
*
-calls
0..*
Bridge
pack_data()
unpack_data()
forward_message()
transmit_message()
*
-uses API
Server
initialize()
enter_main_loop()
run_service()
use_Broker_API()

Broker Architectural Pattern



- Hub-and-spoke architecture - broker acts as a messaging hub
  - Senders and receivers connect as spokes .
  - Connections to the broker are via ports that are associated with a specific message format
- Broker Performs message routing - delivery path can be hard coded or depend on values in the input message
- Performs message transformation : The broker transforms the source message type to destination message type

Broker – Advantages

- Availability – can replicate brokers (clustering as in messaging and publish–subscribe)
- Failure handling - brokers have typed input ports, they validate and discard any messages with wrong format.
  - replicated brokers - senders can failover to a live broker
- Modifiability - Brokers separate the transformation and message routing logic from the senders and receivers. This enhances modifiability, as changes to transformation and routing logic can be made without affecting senders or receivers
- Performance - Brokers can potentially become a bottleneck, especially if high message volumes / complex transformations
- Scalability – Can add n brokers so scaling possible
- Reuseability – components need not be interdependent

Broker Disadvantages

- Overall system performance will not be as high as that of a system
  with direct client/server connections.
  - Tradeoff with benefit of how easy it is to add new services + portability,
    flexibility, and changeability of the broker architecture.
  - Single point of failure into the architecture – unless redundancy is built in

MVC

- Model View Separation Principle

1. Do not connect or couple non-UI objects directly to UI objects.

   – E.g. Sale software object should not reference to Java Swing JFrame object
     - Why – spoils reusability! Sale should not be coupled with the UI

2. Do not put application logic (e.g. tax calculation) in the UI object methods
   – UI objects should only be concerned with UI functions

- Principle is key in Model-View-Controller pattern
  – Model = Domain Layer (Data)
  – View = UI Layer (Windows, Buttons, layout etc)
  – Controller =Application Layer (handling view requests – mouse events etc)

Model-View Separation – Why?

- Allow focus on domain processes (not UI)
- Separate development of model and UI
- Minimise effect of changes to UI
- New views (UI) can be added with existing domain layer
- Multiple views of same domain
- Easy porting to another UI framework (e.g. client-server to web platform)

Example MVC in J2EE (Java EE)

- Model = JavaBean that exposes data.
  - Model should not know how to retrieve data from business objects
  - Exposes properties that enable its state to be initialised by a controller
- View = component used to display the data in a model e.g. a JSP page
  - Each view does one thing – e.g. display a certain set of objects
  - Each view replaceable with another view – e.g. to display the same model differently – e.g. as text, as table etc
- Controller = Java class to handle incoming requests.
  - Interacts with business objects, builds models, forwards request to appropriate view

MVC CRC Cards

Class Model	***Collaborators***
Responsibilities	• View
• Provide functional core of an application	• Controller
• Register views and controller interest in model data	
• Notify registered components about data changes	

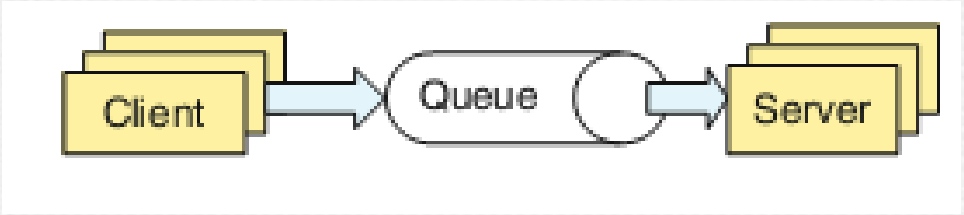
Class View	***Collaborators***
Responsibilities	• Controller
• Creates and initializes its controller	• Model
• Displays information to the user	
• Updates itself when new data arrives from model	
• Retrieves data from the model	

Class Controller	***Collaborators***
Responsibilities	• View
• Accepts user input as events	• Model
• Translates events into requests for the model or display request for the view	
• Updates itself when new data arrives from the Model	

MVC Advantages

- Use the same data to supply multiple views
- Changes to the data in the underlying model are reflected in all the views automatically
- Easy to change the view and controller elements of the system without changing the data model = flexibility
- Because the UI code is independent of the model, when you need to make major changes in the UI section, the underlying data doesn't need to change
- >1 View - views don't interact. As a result, you can change an individual view without having to make changes in the other views
- MVC architectures can be used as frameworks to be used and extended in other situations. The three components are related yet independent, which simplifies maintenance and evolution.

Messaging Architectural Pattern



- Asynchronous communications :
  - Clients send requests to the queue
  - message is stored in queue until an application removes it.
  - After the client has written the message to the queue, it continues without waiting for the message to be removed .
- Configurable QoS :
  - high-speed, reduced reliablility versus slower more reliable delivery
- Loose coupling - no direct binding between clients and servers.
  - client is oblivious to which server receives the message .
  - server is oblivious as to which client the message cam e from.

Messaging Architectural Pattern cont.

- Especially appropriate when the client does not need an immediate response directly after sending a request.
  - E.g. Email system – client places an email on a queue for processing. The server will at some stage in the future remove the message and send the email using a mail server.
  - Applications that can divide processing of a request into a number of discrete steps, connected by queues, are a basic extension of the simple messaging pattern.
  - identical to the “Pipe and Filter” pattern
- Messaging also provides a resilient solution for applications in which connectivity to a server application is transient, either due to network or server unreliability.
  - messages are held in the queue until the server connects and removes it.

Messaging – quality attributes

- Availability - Physical queues with the same logical name can be replicated across different messaging server instances. When one fails, clients can send messages to replica queues
- Failure handling - If a client is communicating with a queue that fails, it can find a replica queue and post the message there
- Modifiability - inherently loose coupling between client and server.
  - Changes to messages format from clients problematic (Solve by self-describing, discoverable message formats.)
- Performance – Good - thousands of messages per second.
  - Reliability – performance trade-off. Non-reliable messaging is faster!
- Scalability - Queues can replicated across clusters of messaging servers (single or multiple server machines)

Publish Subscribe Architectural Pattern

- N:M messaging - Published messages are sent to all subscribers who are registered with the topic.
- Many publishers can publish on the same topic, and many subscribers can listen to the same topic.
- Configurable QoS -
  - non-reliable and reliable messaging,
  - communication mechanism may be
    - point -to-point - distinct message for ever y subscriber on a topic
    - broad cast/multicast - one message which ever y subscriber
- Loose Coupling - no direct binding between publishers and subscribers.

Publish Subscribe – quality attributes

- Availability - Topics with the same logical name can be replicated across different server instances managed as a cluster.
- Failure handling - publisher can find a live replica server if its topic server fails and send the message there
- Modifiability - inherently loosely coupled
  - Add new publishers and subscribers easily
- Performance - thousands of messages per second (as usual non-reliable messaging faster than reliable.
- Scalability - Topics can be replicated across clusters of servers hosted on a single or multiple server machines. Easy to scale. (multicast/broadcast solutions scale better than their point-to-point counterparts)

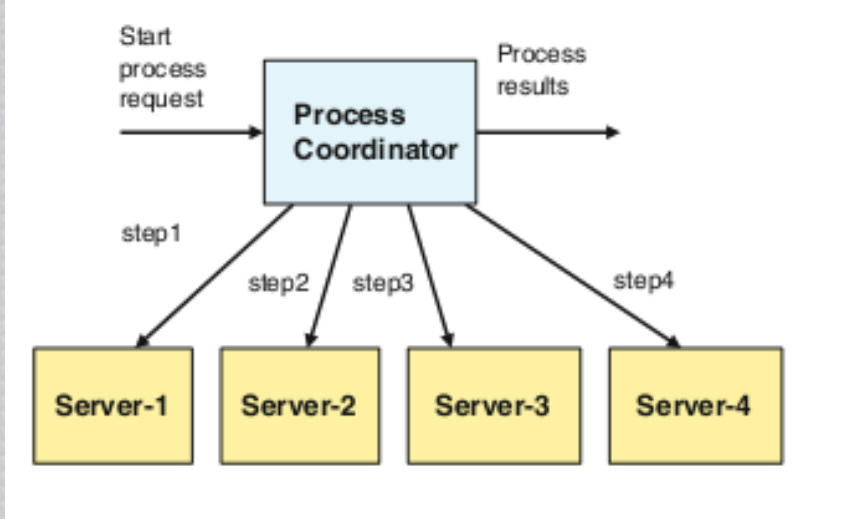
Event-based architectural style

- Uses implicit invocation ie services do not talk to each other
- Event = triggers / changes
  - alerts
  - messages,
  - user inputs
  - outputs from other services
- Uses an event bus =- connector between event generators and consumers
- Any service/function can be a generator or consumer
- Register for notification (cf observer pattern)

Event-based architectural style

- Loose coupling
  - generators don't know who consumes their events
  - consumers **might not** know who triggered them (it depends…)
  - When an event happens – those who need to know are told
  - Makes scaling easier
  - Makes reuse easier
  - Usually asynchronous design – better efficiency
    - BUT possible race condition
    - Possible contention (could use semaphores)

Process Coordinator Architectural Style



- Commonly used to implement complex business processes that must issue requests to several different server components
- Process logic encapsulated in one place ∴ easier to change and monitor process performance.

Process Coordinator Architectural Style

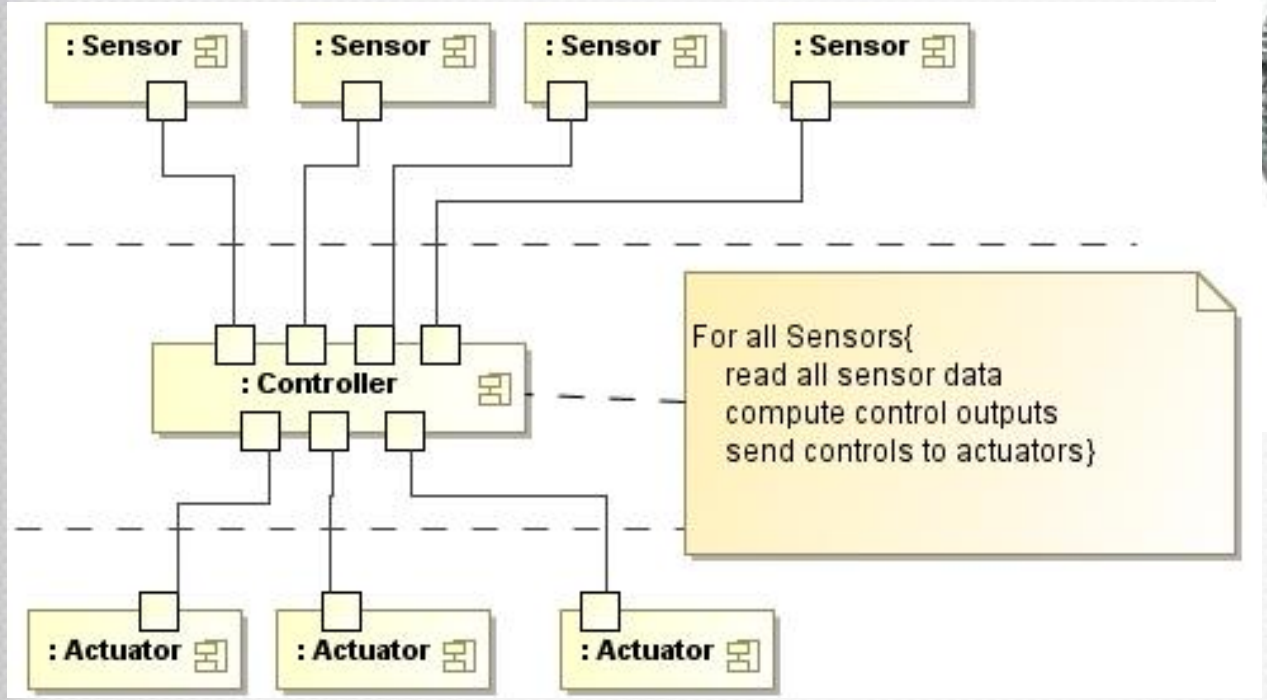
- Process encapsulation : The process coordinator encapsulates the sequence of steps needed to fulfil the business process.
- Coordinator is a single point of definition for the business process, making it easier to understand and modify. It receives a process initiation request, calls the servers in the order defined by the process, and emits the results. (cf Mediator)
- Loose coupling : The server component s are unaware of their role in the overall business process
- Flexible communications : Communications between the coordinator and servers can be synchronous or asynchronous

Process Coordinator quality attributes

- Availability - The coordinator is a single point of failure. Can be replicated
- Failure handling - Failure handling is complex
  - Failure of a later step in the process may require earlier steps to be undone
  - needs careful design to ensure the data on the servers remains consistent
- Modifiability - process definition is encapsulated in the coordinator process. Servers can change their implementation - as long as their external service definition doesn't change
- Performance - coordinator must be able to handle multiple concurrent requests
  - performance of any process will limited by the slowest step
- Scalability - The coordinator can be replicated to scale up and out

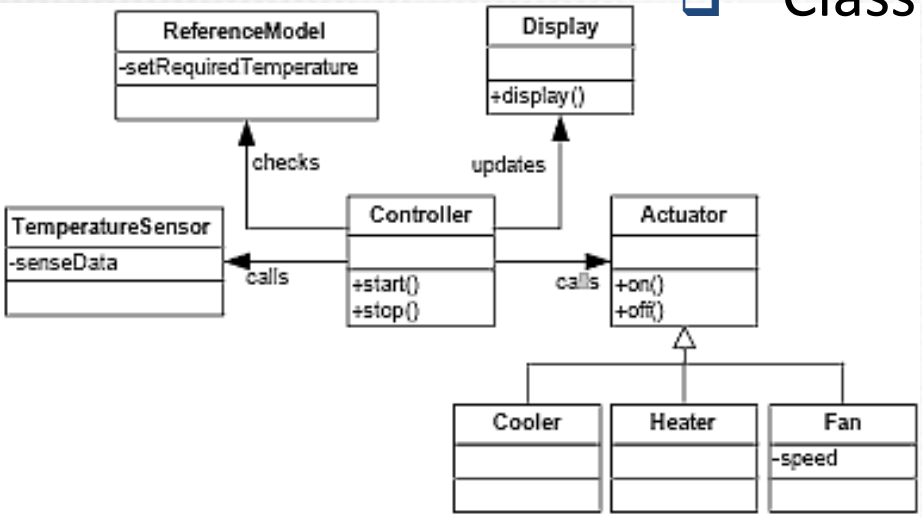
Sensor-Controller-Actuator Architectural Pattern

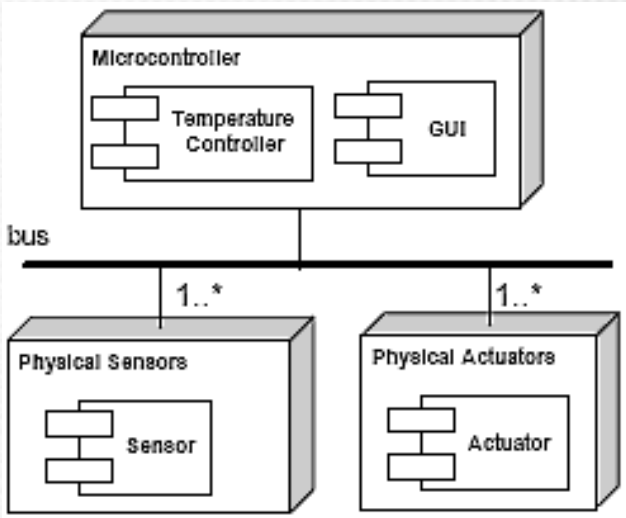
- Aka Sense-Compute-Control



Sensor-Controller-Actuator example

- Climate Controller
  - Class Diagram



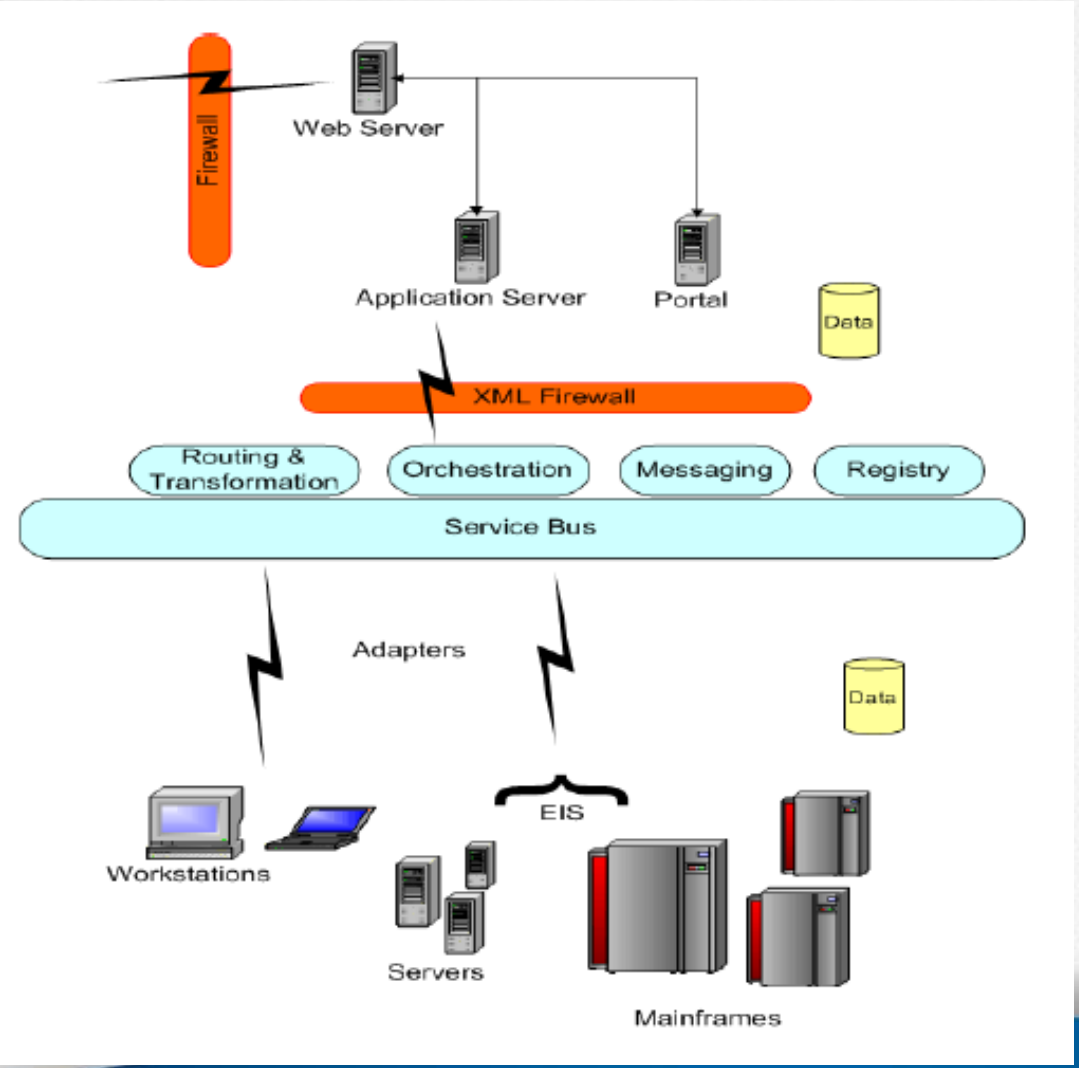


  - Deployment Diagram

Service Oriented Architecture

- Service = unit of business functionality
- Services have a lifecycle - start/stop, migrate, upgrade
- Accessed via some distributed protocol
  - Decentralised (unlike n-tier)
- Service may have metadata e.g. what they offer
- Registry services (WSDL)
- ‘Quality of Service’ guarantees often defined e.g. performance
- Typically stateless (or session mechanisms used)
- Messaging often unreliable – allow resubmission
- Not RPC (RPC has tight coupling)

Example of SOA



SOA needs

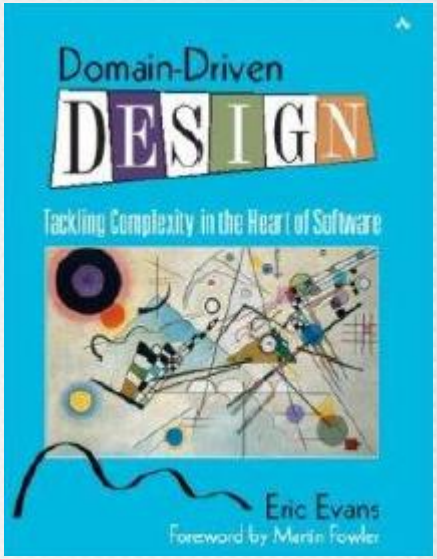
- loosely coupled
- •Registry services to find WSDL
- •Web Support (if using web services)
- •Constraints
- •Typically stateless
- •If stateful session mechanisms needed
- •Must be idempotent
- •Messaging unreliable, must support resubmission
- •Avoid RPC
- •RPC relies on types and tight coupling

Microservices

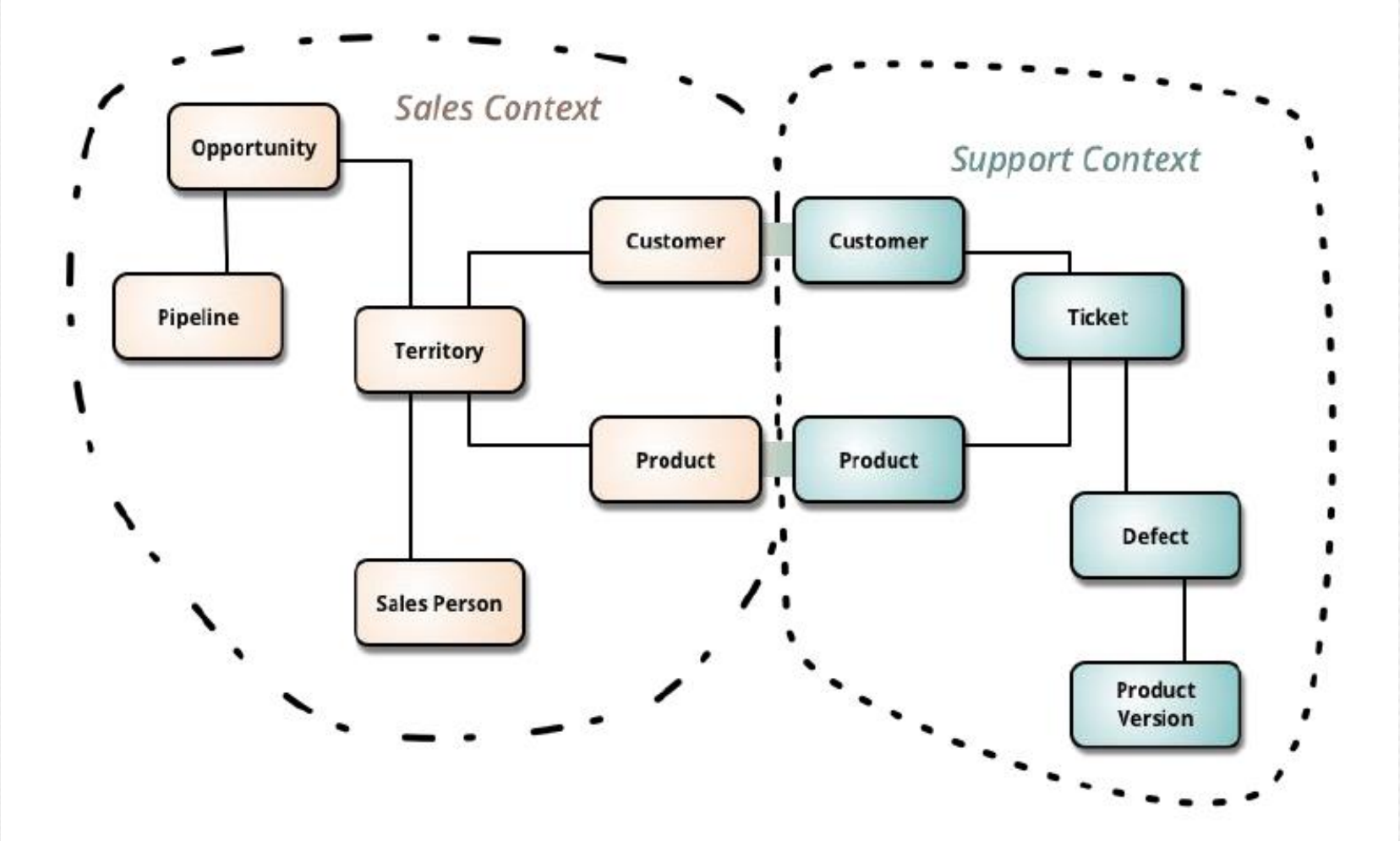
- Microservices = implementation strategy for SOA
- Services are:
  - Small, e.g. developed in Agile sprint
  - High cohesion – e.g. single business task
  - deployable on their own (loose/no coupling)
  - Usually use containers (e.g. docker)
  - Conform to integration standards e.g. RESTful
  - Close relation to DevOps + Continuous Integration + Continuous Delivery
  - Monitoring employed e.g Nagios

Domain Driven Design

- **Domain =** Area of application e.g. accounting, personnel, lab management …
- **Model = h**ow to represent the domain
- **Ubiquitous Language =** a way to model using some language for domain experts
- Key notions:
- Finding classes - e.g. nouns from text
- Finding methods e.g. verbs = methods
- **Bounded Context** to define the scope of the model and its partitions

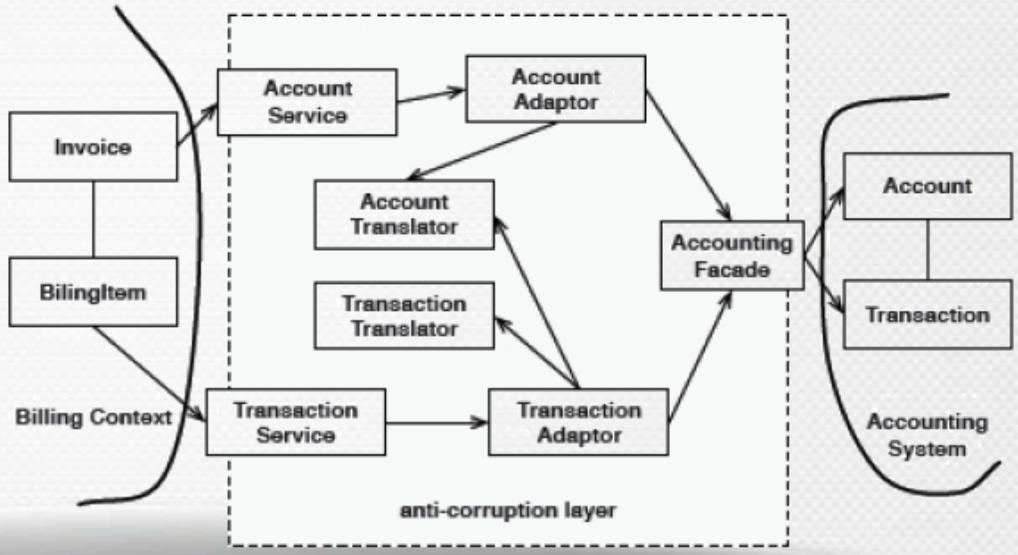


Bounded Contexts



Mapping Contexts

- Choose an architectural style/pattern
- Map Contexts to partitions e.g. layers, repository, etc
- Commonly
  - Shared Kernel is a sub context that appears in many contexts
  - Anti corruption layer



Implementation of Microservices

- Contexts get mapped to 1 or more services
- Services delivered in a (for example) sprint
- TDD
- SCM (e.g. git)
- DevOps – automated build and test
- Implement services in containers (e.g via Docker)
  - Single VM or Cloud instance
- Orchestration – scripts for provisioning policies (Puppet, Chef etc)
- Monitoring – performance, availability etc to meet SLA via a tool (e.g. Nagios)

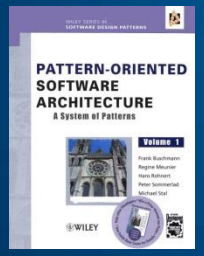
Digital Transformation: CSC4008

Software Architecture

Learning Outcomes

Appreciate the factors in making a choice of software architecture (tactics)

Know a range of Software Architecture Styles and Patterns



Ref: Larman Chapter 29
Software Architecture: Foundations, Theory, and Practice – Taylor et al. Ch.6
L.Bass et al - Software Architecture in Practice