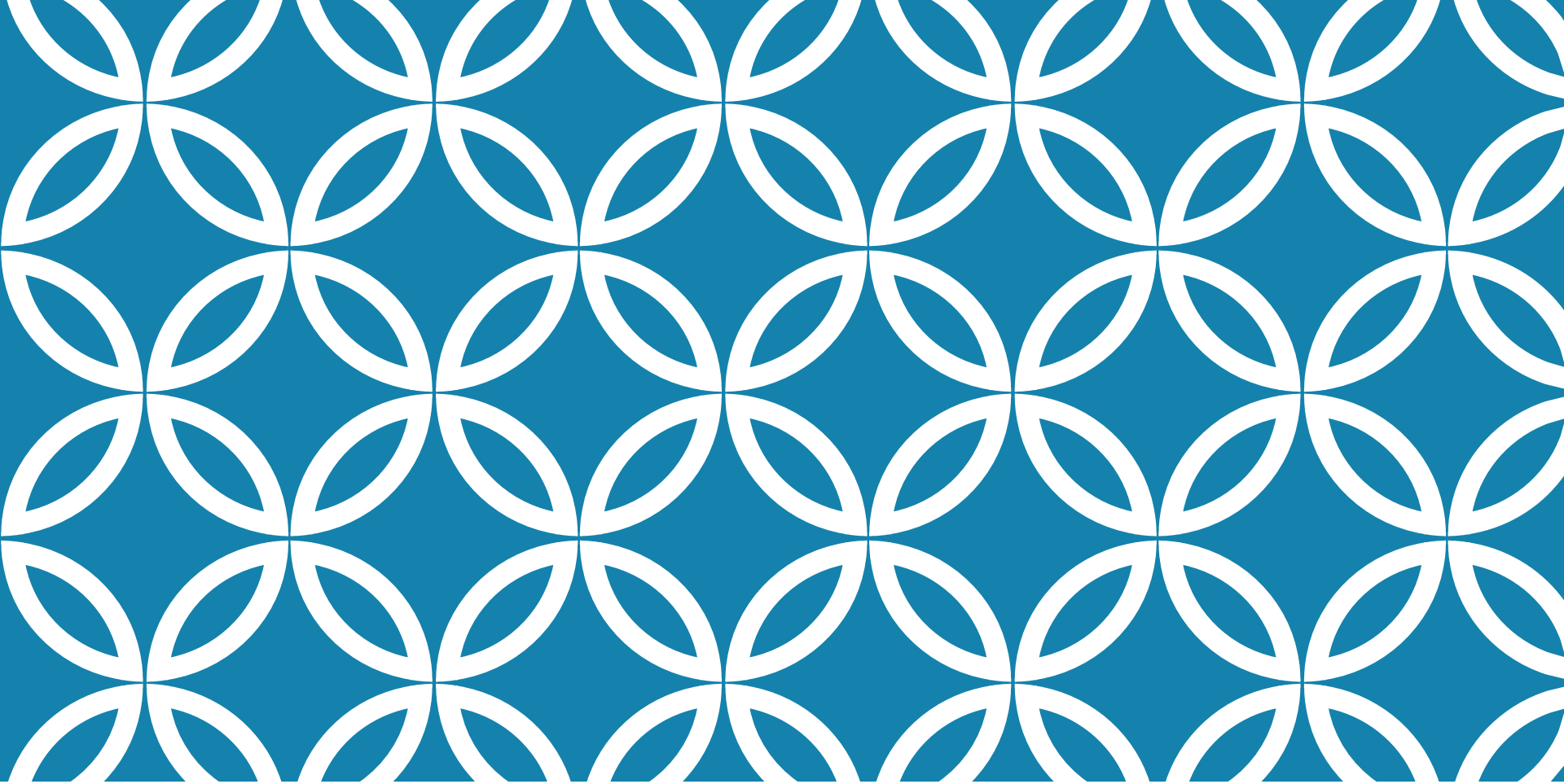


SOFTWARE ARCHITECTURE

System Analysis & Design Course

Sharif University of Technology

IT Systems Application Domain

- IT systems are everywhere
  - Banks
  - Shops
  - Internet sites
- Large, complex, heterogeneous, distributed applications
- Use commercial-of-the-shelf middleware, databases, web servers, application packages
- Major problems are architecture design, technology selection, application and business integration

What is Software Architecture?

- It's about software design
  - All architecture is software design, but not all design is software architecture
  - Part of the design process
- Simply, architecture focuses on 'issues that will be difficult/impossible to change once the system is built'
  - Quality attributes like security, performance
  - Non-functional requirements like cost, deployment hardware
  - More on these later in this session

Defintions

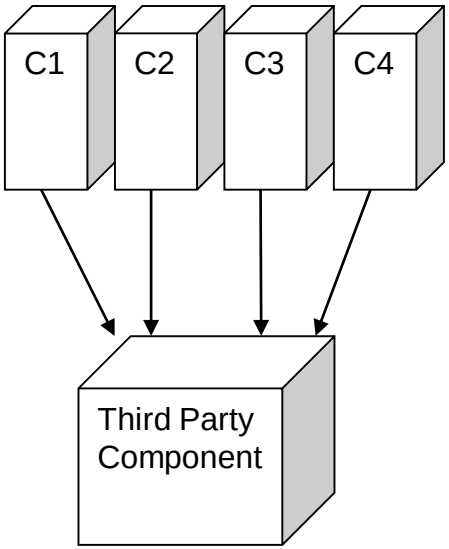
- *"Architecture is the **fundamental organization** of a system, embodied in its **components**, their **relationships** to **each other** and the **environment**, and the principles governing its **design and evolution**."*

Architecture Defines Structure

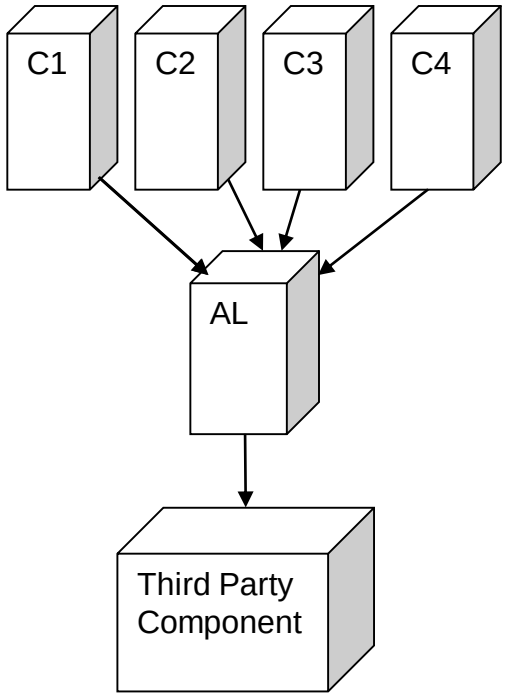
- Decomposition of system in to components/modules/subsystems
- Architecture defines:
  - Component interfaces
    - What a component can do
  - Component communications and dependencies
    - How components communicate
  - Component responsibilities
    - Precisely what a component will do when you ask it

Structure and Dependencies

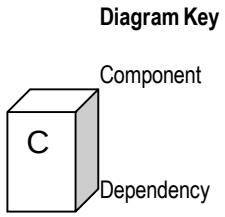
- Excessive component dependencies are bad!
- Key architecture issue
  - Identifying components that may change
  - Reduce direct dependencies on these components
- Creates more modifiable systems



Four components are directly dependent on a third party component. If the third party component is replaced with a new component with a different interface, changes to each component are likely.



Only the AL (abstraction layer) component is directly dependent on the third party component. If the third party component is replaced, changes are restricted to the AL component only



Architecture Specifies Component Communication

- Communication involves:
  - Data passing mechanisms, e.g.:
    - Function call
    - Remote method invocation
    - Asynchronous message
  - Control flow
    - Flow of messages between components to achieve required functionality
    - Sequential
    - Concurrent/parallel
    - Synchronization

Architecture Patterns/Styles

- Patterns catalogue successfully used structures that facilitate certain kinds of component communication
  - client-server
  - Message broker
  - Pipeline
- Patterns have well-known characteristics appropriate for particular types of requirements
- Patterns are very useful things …
  - Reusable architectural blueprints
  - Help efficiently communicate a design
  - Large systems comprise a number of individual patterns
  - “Patterns and Styles are the same thing – the patterns people won” [anonymous SEI member]

Architecture addresses NFRs

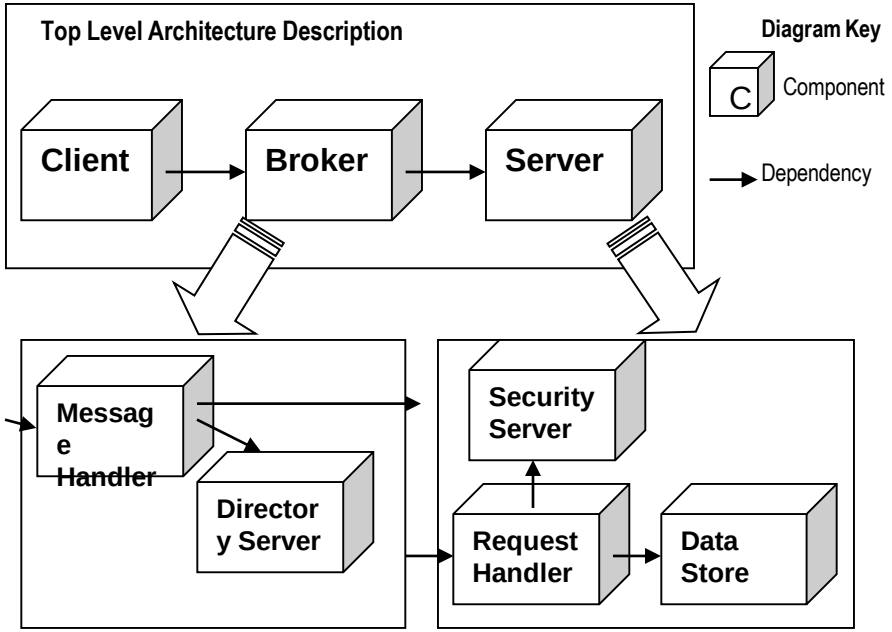
- Non-functional requirements (NFRs) define ‘how’ a system works
- NFRs rarely captured in functional requirements
  - Aka. architecture requirements
  - Must be elicited by architect
- NFRs include:
  - Technical constraints
  - Business constraints
  - Quality attributes

Architecture is an Abstraction

- Architecture provides an abstract view of a design
  - Hides complexity of design
  - May or may not be a direct mapping between architecture elements and software elements
- Example – A Marketecture
  - informal depiction of system's structure and interactions.
  - portray the design philosophies embodied in the architecture
- Every system should have a marketecture:
  - Easy to understand
  - Helps discussion during design, build, review, sales (!) process

Decomposition

- Hierarchical decomposition is a powerful abstraction mechanism
  - Partitions design
  - Allocate components to development teams



Architecture Views

- A software architecture represents a complex design artifact
- Many possible ‘views’ of the architecture
  - Cf. with buildings – floor plan, external, electrical, plumbing, air-conditioning

Philippe Krutchen - *4+1 View Model*

- **Logical view:** describes architecturally significant elements of the architecture and the relationships between them.
- **Process view:** describes the concurrency and communications elements of an architecture.
- **Physical view:** depicts how the major processes and components are mapped on to the applications hardware.
- **Development view:** captures the internal organization of the software components as held in e.g. a configuration management tool.
- **Architecture use cases:** capture the requirements for the architecture; related to more than one particular view

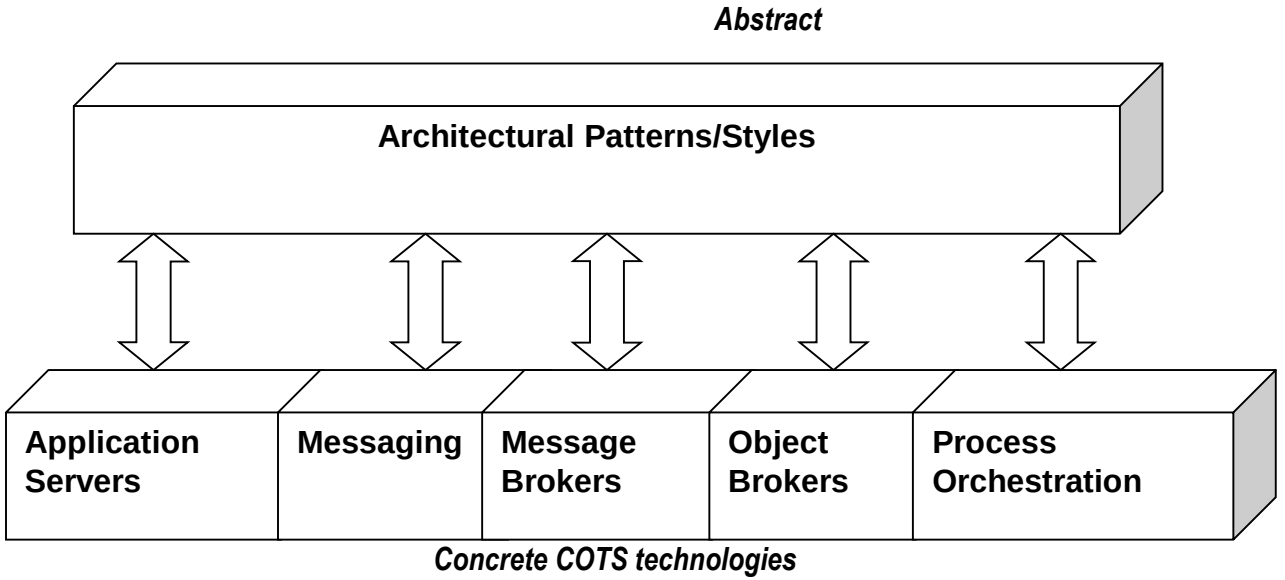
What does an Architect do?

- Many responsibilities:
  - Liaison with stakeholders
  - Technology knowledge
  - Software engineering
  - Risk managements

Architectures and Technologies

- Architects reduce risks by using proven design patterns
- Must map abstract design pattern to concrete implementation
- Software vendors have created (COTS) technologies that explicitly support widely used patterns
  - Makes implementation of patterns easier
  - Reduces risk if technology is well built

Architectures and Technologies



- Each technology has multiple vendors/open source versions
- Architects need to choose technology wisely
  - Proof of concept prototype
  - Detailed feature evaluation

What are Quality Attributes

- Often know as –ilities
  - Reliability
  - Availability
  - Portability
  - Scalability
  - Performance (!)
- Part of a system's NFRs
  - "how" the system achieves its functional requirements

Quality Attribute Specification

- QA's must be concrete
- But what about testable?
  - Test scalability by installing system on 10K desktops?
- Often careful analysis of a proposed solution is all that is possible
- "It's all talk until the code runs"

Performance

- Many examples of poor performance in enterprise applications
- Performance requires a:
  - Metric of amount of work performed in unit time
  - Deadline that must be met
- Enterprise applications often have strict performance requirements, e.g.
  - 1000 transactions per second
  - 3 second average latency for a request

Scalability

- *"How well a solution to some problem will work when the size of the problem increases."*
- 4 common scalability issues in IT systems:
  - Request load
  - Connections
  - Data size
  - Deployments

Modifiability

- Modifications to a software system during its lifetime are a fact of life.
- Modifiable systems are easier to change/evolve
- Modifiability should be assessed in context of how a system is likely to change
  - No need to facilitate changes that are highly unlikely to occur
  - Over-engineering!

Security

- Difficult, specialized quality attribute:
  - Lots of technology available
  - Requires deep knowledge of approaches and solutions
- Security is a multi-faceted quality …

Security

- **Authentication:** Applications can verify the identity of their users and other applications with which they communicate.
- **Authorization:** Authenticated users and applications have defined access rights to the resources of the system.
- **Encryption:** The messages sent to/from the application are encrypted.
- **Integrity:** This ensures the contents of a message 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 their participation in the message exchange.

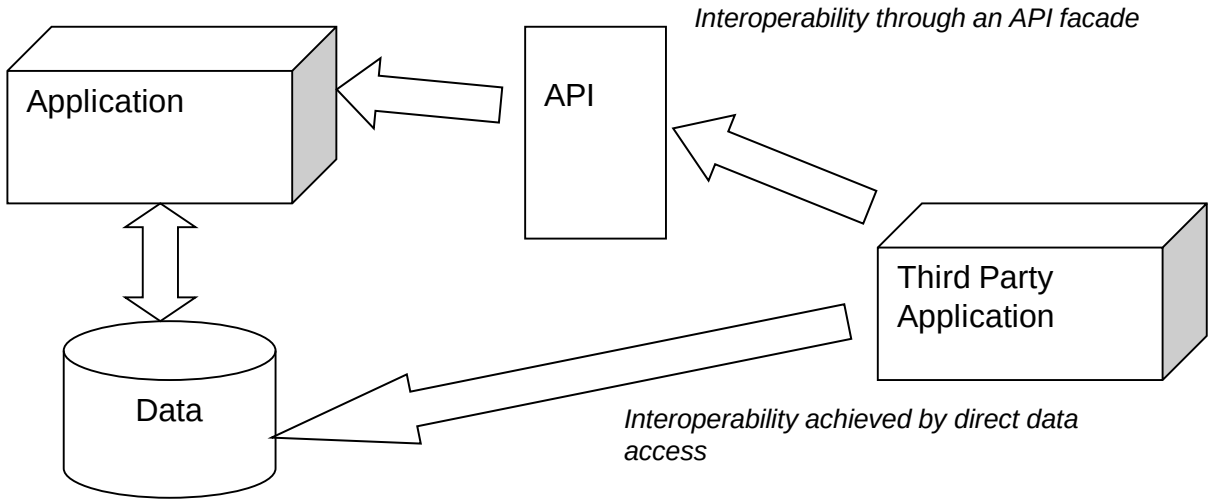
Availability

- Key requirement for most IT applications
- Measured by the proportion of the required time it is useable. E.g.
  - 100% available during business hours
  - No more than 2 hours scheduled downtime per week
  - 24x7x52 (100% availability)
- Related to an application's reliability
  - Unreliable applications suffer poor availability

Integration

- ease with which an application can be incorporated into a broader application context
  - Use component in ways that the designer did not originally anticipate
- Typically achieved by:
  - Programmatic APIs
  - Data integration

Integration Strategies



- Data – expose application data for access by other components
- API – offers services to read/write application data through an abstracted interface
- Each has strengths and weaknesses …

Misc. Quality Attributes

- Portability
  - Can an application be easily executed on a different software/hardware platform to the one it has been developed for?
- Testability
  - How easy or difficult is an application to test?
- Supportability
  - How easy an application is to support once it is deployed?

Design Trade-offs

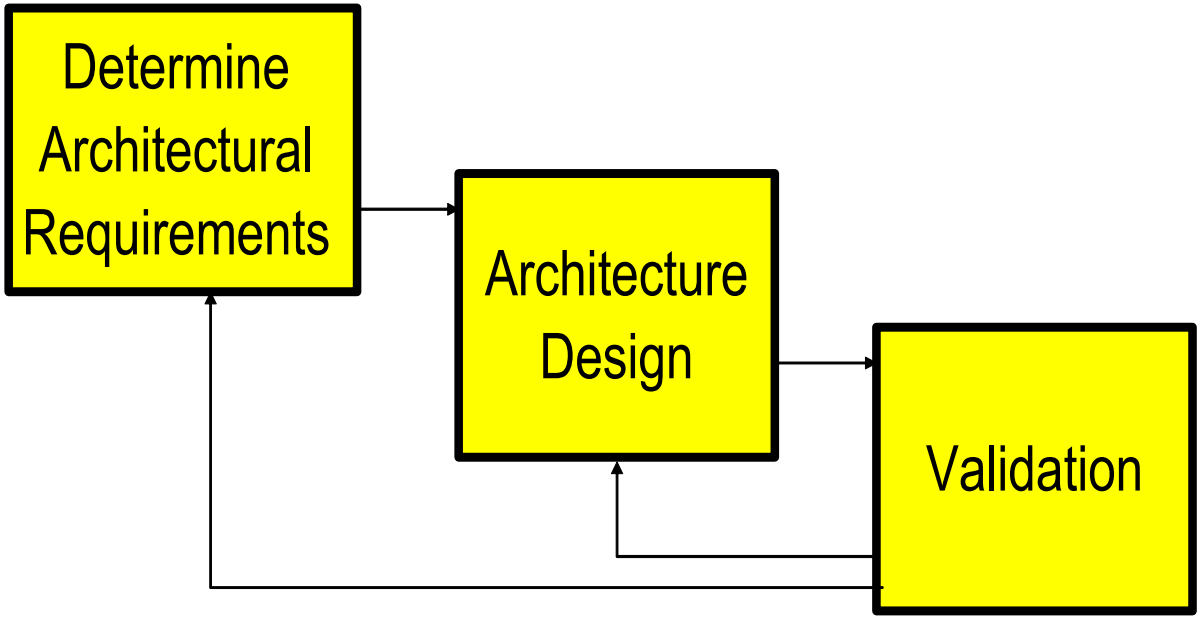
- QAs are rarely orthogonal
  - They interact, affect each other
  - highly secure system may be difficult to integrate
  - highly available application may trade-off lower performance for greater availability
  - high performance application may be tied to a given platform, and hence not be easily portable
- Architects must create solutions that makes sensible design compromises
  - not possible to fully satisfy all competing requirements
  - Must satisfy all stakeholder needs
  - This is the difficult bit!

A Software Architecture Process

- Architects must be versatile:
  - **Work with the requirements team:** The architect plays an important role in requirements gathering by understanding the overall systems needs and ensuring that the appropriate quality attributes are explicit and understood.
  - **Work with various application stakeholders:** Architects play a pivotal liaison role by making sure all the application's stakeholder needs are understood and incorporated into the design.
  - **Lead the technical design team:** Defining the application architecture is a design activity.
  - **Work with the project management:** Planning, estimates, budgets, schedules

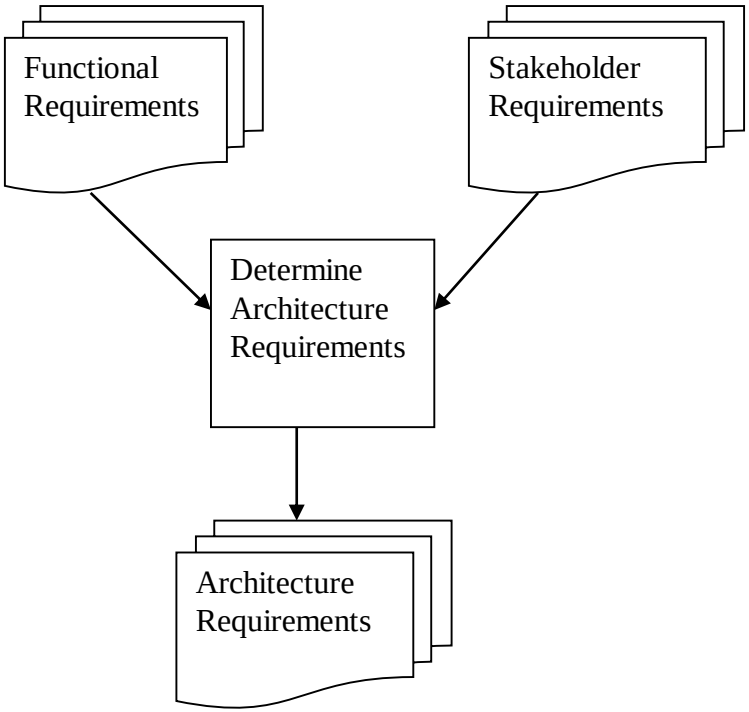
An Architecture Process

- Highly iterative
- Can scale to small/large projects



Determine Architectural Requirements

- Sometime called:
  - architecturally significant requirements
  - architecture use cases
- essentially the quality and non-functional requirements for a system.



Examples

- A typical architecture requirement :
  - *“Communications between components must be guaranteed to succeed with no message loss”*
- Some architecture requirements are constraints:
  - “*The system must use the existing IIS-based web server and use Active Server Page to process web requests*”
- Constraints impose restrictions on the architecture and are (almost always) non-negotiable.
- They limit the range of design choices an architect can make.

Quality Attribute 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 512MB memory.
Usability	The user interface component must run in an Internet browser to support remote users.
Availability	The system must run 24x7x365, with overall availability of 0.99.
Reliability	No message loss is allowed, and all message delivery outcomes must be known with 30 seconds
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.

Constraints

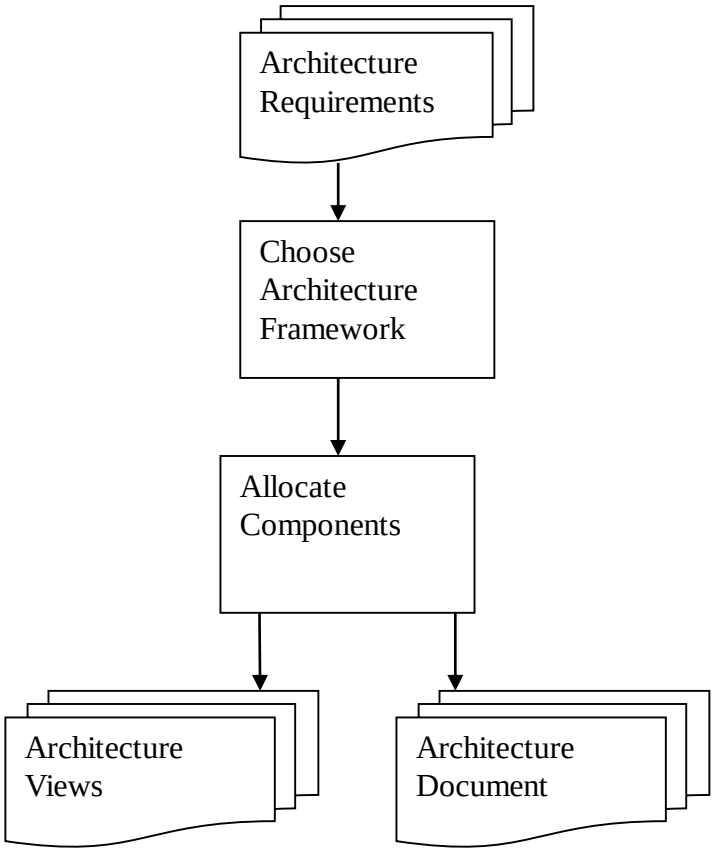
Constraint	Architecture Requirement
Business	The technology must run as a plug-in for MS BizTalk, as we want to sell this to Microsoft.
Development	The system must be written in Java so that we can use existing development staff.
Schedule	The first version of this product must be delivered within six months.
Business	We want to work closely with and get more development funding from *MegaHugeTech Corp*, so we need to use their technology in our application.

Priorities

- All requirements are not equal
  - **High:** the application must support this requirement.
  - **Medium:** this requirement will need to be supported at some stage
  - **Low:** this is part of the requirements wish list.
- Tricky in face of conflicts, eg:
  - Reusability of components in the solution versus rapid time-to-market. Making components generalized and reusable always takes more time and effort.
  - Minimal expenditure on COTS products versus reduced development effort/cost. COTS products mean you have to develop less code, but they cost money.
- It's design – not meant to be easy!

Architecture Design

- Design steps are iterative
- Risk identification is a crucial output of the design

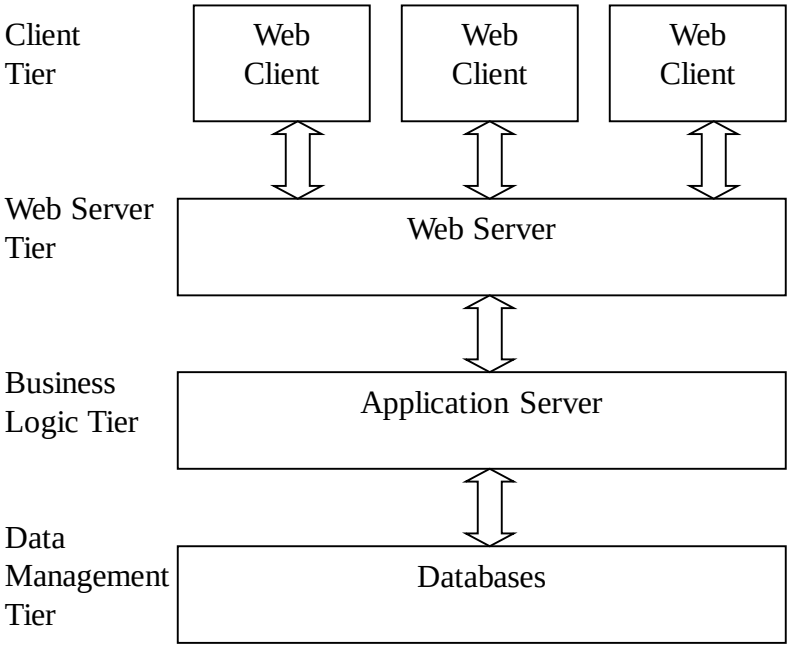


Choosing the Architecture Framework

- Choose a architecture pattern/patterns that suit requirements
  - No magic formula
  - Analyze requirements and quality attributed supported by each pattern
- Complex architectures require creative blending of multiple patterns.

N-Tier Client Server Pattern

- **Separation of concerns:** Presentation, business and data handling logic are clearly partitioned in different tiers.
- **Synchronous communications:** Communications between tiers is synchronous request-reply. Each tier waits for a response from the other tier before proceeding.
- **Flexible deployment:** There are no restrictions on how a multi-tier application is deployed. All tiers could run on the same machine, or each tier may be deployed on its own machine.

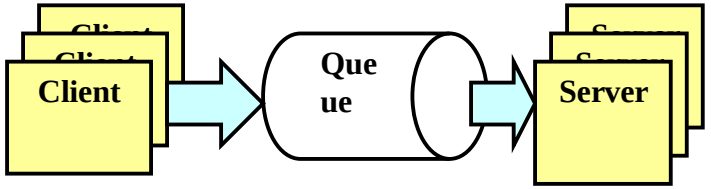


N-Tier Client Server – Quality Attribute Analysis

Quality Attribute	Issues
Availability	Servers in each tier can be replicated, so that if one fails, others remain available. Overall the application will provide a lower quality of service until the failed server is restored.
Failure handling	If a client is communicating with a server that fails, most web and application servers implement transparent failover. This means a client request is, without its knowledge, redirected to a live replica server that can satisfy the request.
Modifiability	Separation of concerns enhances modifiability, as the presentation, business and data management logic are all clearly encapsulated. Each can have its internal logic modified in many cases without changes rippling into other tiers.
Performance	This architecture has proven high performance. Key issues to consider are the amount of concurrent threads supported in each server, the speed of connections between tiers and the amount of data that is transferred. As always with distributed systems, it makes sense to minimize the calls needed between tiers to fulfill each request.
Scalability	As servers in each tier can be replicated, and multiple server instances run on the same or different servers, the architecture scales out and up well. In practice, the data management tier often becomes a bottleneck on the capacity of a system.

Messaging Pattern

- **Asynchronous communications:** Clients send requests to the queue, where the message is stored until an application removes it. **Configurable QoS:** The queue can be configured for high-speed, non-reliable or slower, reliable delivery. Queue operations can be coordinated with database transactions.
- **Loose coupling:** There is no direct binding between clients and servers.

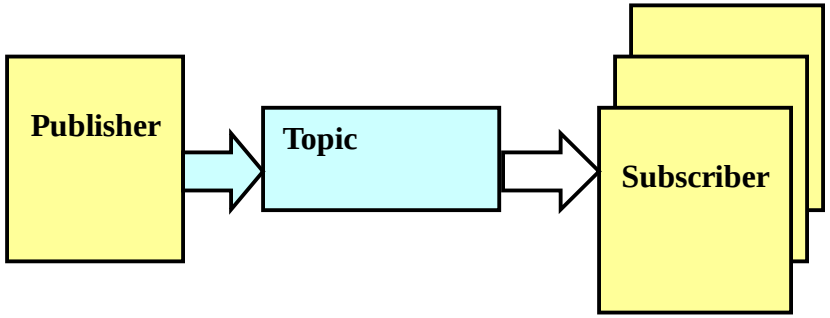


Messaging – Quality Attribute Analysis

Quality Attribute	Issues
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	Messaging is inherently loosely coupled, and this promotes high modifiability as clients and servers are not directly bound through an interface. Changes to the format of messages sent by clients may cause changes to the server implementations. Self-describing, discoverable message formats can help reduce this dependency on message formats.
Performance	Message queuing technology can deliver thousands of messages per second. Non-reliable messaging is faster than reliable, with the difference dependent of the quality of the messaging technology used.
Scalability	Queues can be hosted on the communicating endpoints, or be replicated across clusters of messaging servers hosted on a single or multiple server machines. This makes messaging a highly scalable solution.

Publish-Subscribe Pattern

- **Many-to-Many messaging:** Published messages are sent to all subscribers who are registered with the topic.
- **Configurable QoS:** In addition to non-reliable and reliable messaging, the underlying communication mechanism may be point-to-point or broadcast/multicast.
- **Loose Coupling:** As with messaging, there is no direct binding between publishers and subscribers.

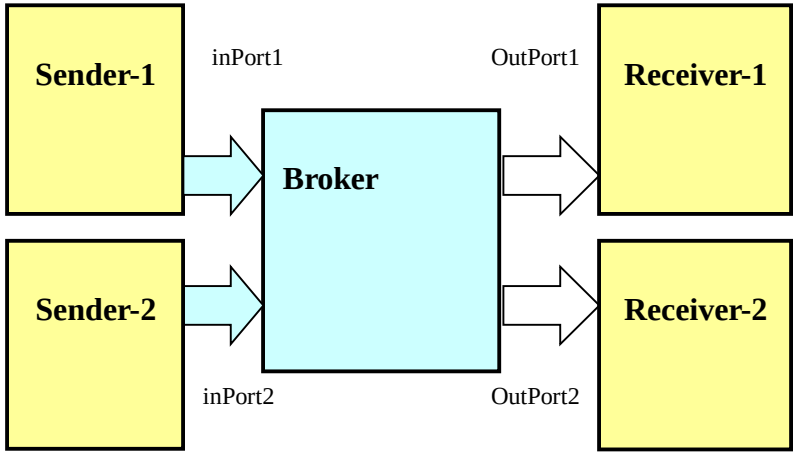


Publish-Subscribe – Quality Attribute Analysis

Quality Attribute	Issues
Availability	Topics with the same logical name can be replicated across different server instances managed as a cluster. When one fails, publishers send messages to replica queues.
Failure handling	If a publisher is communicating with a topic hosted by a server that fails, it can find a live replica server and send the message there.
Modifiability	Publish-subscribe is inherently loosely coupled, and this promotes high modifiability. New publishers and subscribers can be added to the system without change to the architecture or configuration. Changes to the format of messages published may cause changes to the subscriber implementations.
Performance	Publish-subscribe can deliver thousands of messages per second, with non-reliable messaging faster than reliable. If a publish-subscribe broker supports multicast/broadcast, it will deliver multiple messages in a more uniform time to each subscriber.
Scalability	Topics can be replicated across clusters of servers hosted on a single or multiple server machines. Clusters of server can scale to provide very high message volume throughput. Also, multicast/broadcast solutions scale better than their point-to-point counterparts.

Broker Pattern

- **Hub-and-spoke architecture:** The broker acts as a messaging hub, and senders and receivers connect as spokes.
- **Performs message routing:** The broker embeds processing logic to deliver a message received on an input port to an output port.
- **Performs message transformation:** The broker logic transforms the source message type received on the input port to the destination message type required on the output port.

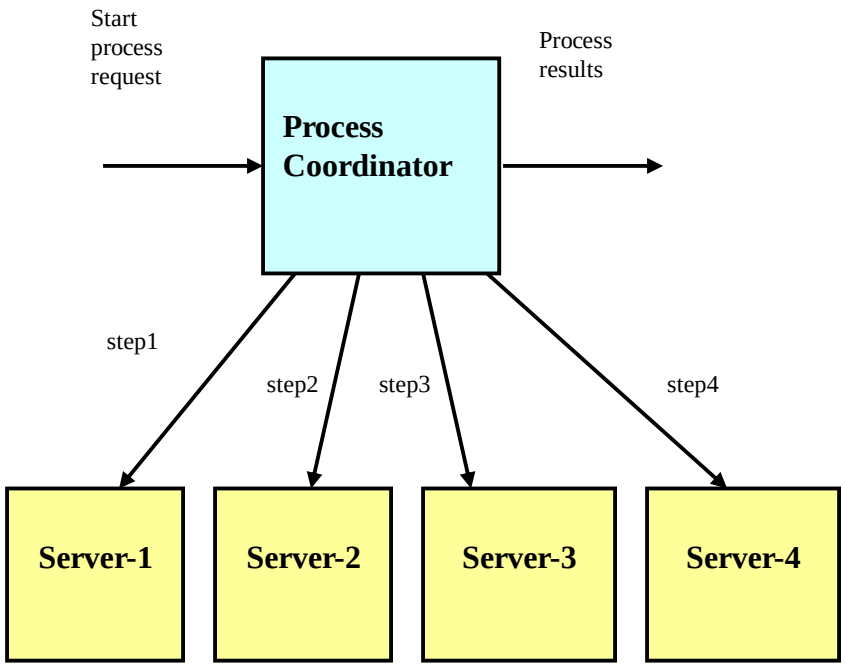


Broker Pattern - Quality Attribute Analysis

Quality Attribute	Issues
Availability	To build high availability architectures, brokers must be replicated. This is typically supported using similar mechanisms to messaging and publish-subscribe server clustering.
Failure handling	As brokers have typed input ports, they validate and discard any messages that are sent in the wrong format. With replicated brokers, senders can fail over to a live broker should one of the replicas fail.
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 they must service high message volumes and execute complex transformation logic. Their throughput is typically lower than simple messaging with reliable delivery.
Scalability	Clustering broker instances makes it possible to construct systems scale to handle high request loads.

Process Coordinator Pattern

- **Process encapsulation:** The process coordinator encapsulates the sequence of steps needed to fulfill the business process. The sequence can be arbitrarily complex.
- **Loose coupling:** The server components are unaware of their role in the overall business process, and of the order of the steps in the process.
- **Flexible communications:** Communications between the coordinator and servers can be synchronous or asynchronous.



Process Coordinator – Quality Attribute Analysis

Quality Attribute	Issues
Availability	The coordinator is a single point of failure. Hence it needs to be replicated to create a high availability solution.
Failure handling	Failure handling is complex, as it can occur at any stage in the business process coordination. Failure of a later step in the process may require earlier steps to be undone using compensating transactions. Handling failures needs careful design to ensure the data maintained by the servers remains consistent.
Modifiability	Process modifiability is enhanced because the process definition is encapsulated in the coordinator process. Servers can change their implementation without affecting the coordinator or other servers, as long as their external service definition doesn't change.
Performance	To achieve high performance, the coordinator must be able to handle multiple concurrent requests and manage the state of each as they progress through the process. Also, the performance of any process will be limited by the slowest step, namely the slowest server in the process.
Scalability	The coordinator can be replicated to scale the application both up and out.

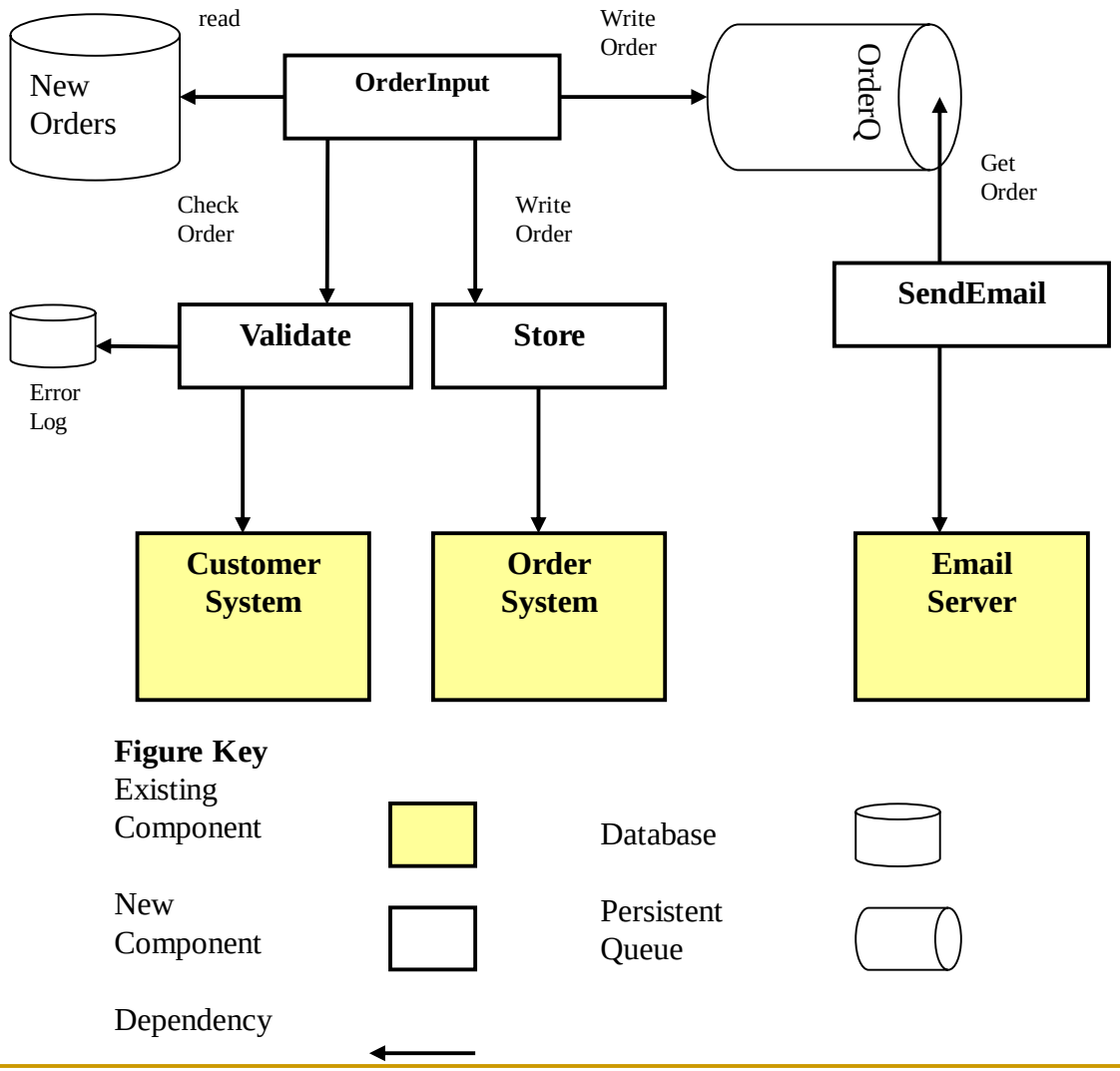
Allocate Components

- Need to:
  - Identify the major application components, and how they plug into the framework.
  - Identify the interface or services that each component supports.
  - Identify the responsibilities of the component, stating what it can be relied upon to do when it receives a request.
  - Identify dependencies between components.
  - Identify partitions in the architecture that are candidates for distribution over servers in a network
    - And independent development

Some Design Guidelines

- Minimize dependencies between components. Strive for a loosely coupled solution in which changes to one component do not ripple through the architecture, propagating across many components.
  - Remember, every time you change something, you have to re-test it.
- Design components that encapsulate a highly “cohesive” set of responsibilities. Cohesion is a measure of how well the parts of a component fit together.
- Isolate dependencies on middleware and any COTS infrastructure technologies.
- Use decomposition to structure components hierarchically.
- Minimize calls between components, as these can prove costly if the components are distributed.

A Simple Design Example



Example Design

- Based on messaging
- Application components are:
  - **OrderInput:** responsible for accessing the new orders database, encapsulating the order processing logic, and writing to the queue.
  - **Validate:** encapsulates the responsibility of interacting with the customer system to carry out validation, and writing to the error logs if an order is invalid.
  - **Store:** responsibility of interacting with the order system to store the order data.
  - **SendEmail:** removes a message from the queue, formats an email message and sends it via an email server. It encapsulates all knowledge of the email format and email server access.
- Clear responsibilities and dependencies

Architecture Validation

- Aim of the validation phase is to increase confidence of the design team that the architecture is fit for purpose.
- The validation has to be achieved within the project constraints of time and budget
  - The trick is to be as rigorous and efficient as possible.
- Validating an architecture design poses tough challenges.
  - ‘coz it’s a design that can’t be executed or tested
  - consists of new and COTS components that have to be integrated
- Two main techniques:
  1. manual testing of the architecture using test scenarios.
  2. construction of a prototype that creates a simple archetype of the desired application
- aim of both is to identify potential flaws in the design so that they can be improved before implementation commences.
  - Cheaper to fix before built

Scenarios

- Part of SEI's ATAM work
- Involves defining:
  - some kind of stimulus that will have an impact on the architecture.
  - working out how the architecture responds to this stimulus.
- If the response is desirable, then a scenario is deemed to be satisfied by the architecture.
- If the response is undesirable, or hard to quantify, then a flaw or at least an area of risk in the architecture may have been uncovered.

Scenario Examples

Quality Attribute	Stimulus	Response
Availability	The network connection to the message consumers fails.	Messages are stored on the MOM server until the connection is restored. Messages will only be lost if the server fails before the connection comes back up.
Modifiability	A new set of data analysis components must be made available in the application.	The application needs to be rebuilt with the new libraries, and the all configuration files must be updated on every desktop to make the new components visible in the GUI toolbox.
Security	No requests are received on a user session for ten minutes.	The system treats this session as potentially insecure and invalidates the security credentials associated with the session. The user must logon again to connect to the application.
Modifiability	The supplier of the transformation engine goes out of business.	A new transformation engine must be purchased. The abstract service layer that wraps the transformation engine component must be re-implemented to support the new engine. Client components are unaffected as they only use the abstract service layer.
Scalability	The concurrent user request load doubles during the 3 week enrollment period.	The application server is scaled out on a two machine cluster to handle the increased request load.

Prototyping

- Scenarios can't address everything:
  - *"On Friday afternoon, orders must be processed before close-of-business to ensure delivery by Monday. Five thousand orders arrive through various channels (Web/Call centre/business partners) five minutes before close-of-business."*
- Only one way – build something!
  - **Proof-of-concept prototype:** Can the architecture as designed be built in a way that can satisfy the requirements?
  - **Proof-of-technology prototype :** Does the technology (middleware, integrated applications, libraries, etc) selected to implement the application behave as expected?

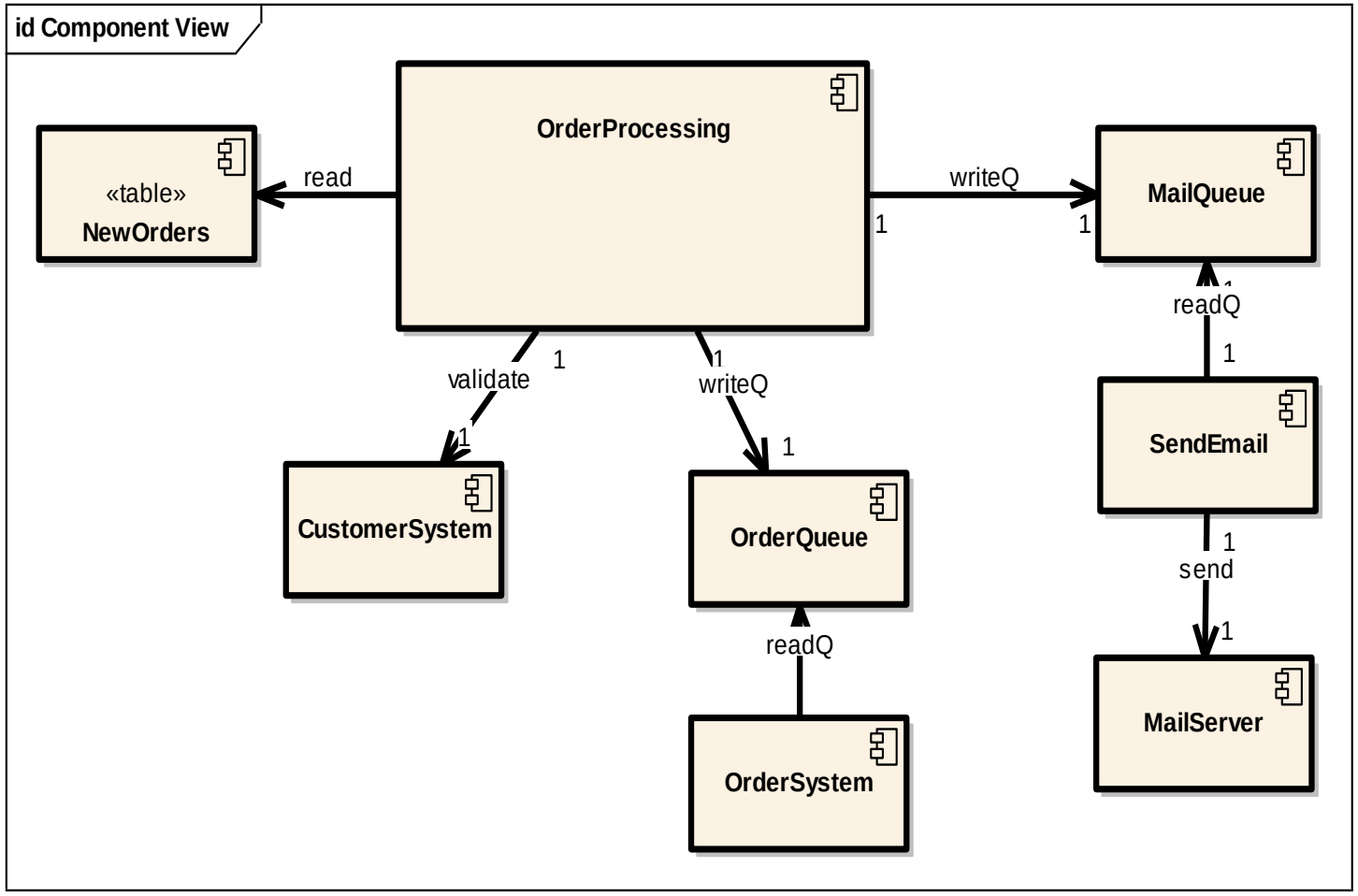
Prototyping Strategy

- Build minimal system required to validate architecture, eg:
- An existing application shows that the queue and email systems are capable of supporting five thousand messages in five minutes
- So:
  - Write a test program that calls the *Customer System* validation APIs five thousand times, and time how long this takes.
  - Write a test program that calls the *Order System* store APIs five thousand times, and time how long this takes.

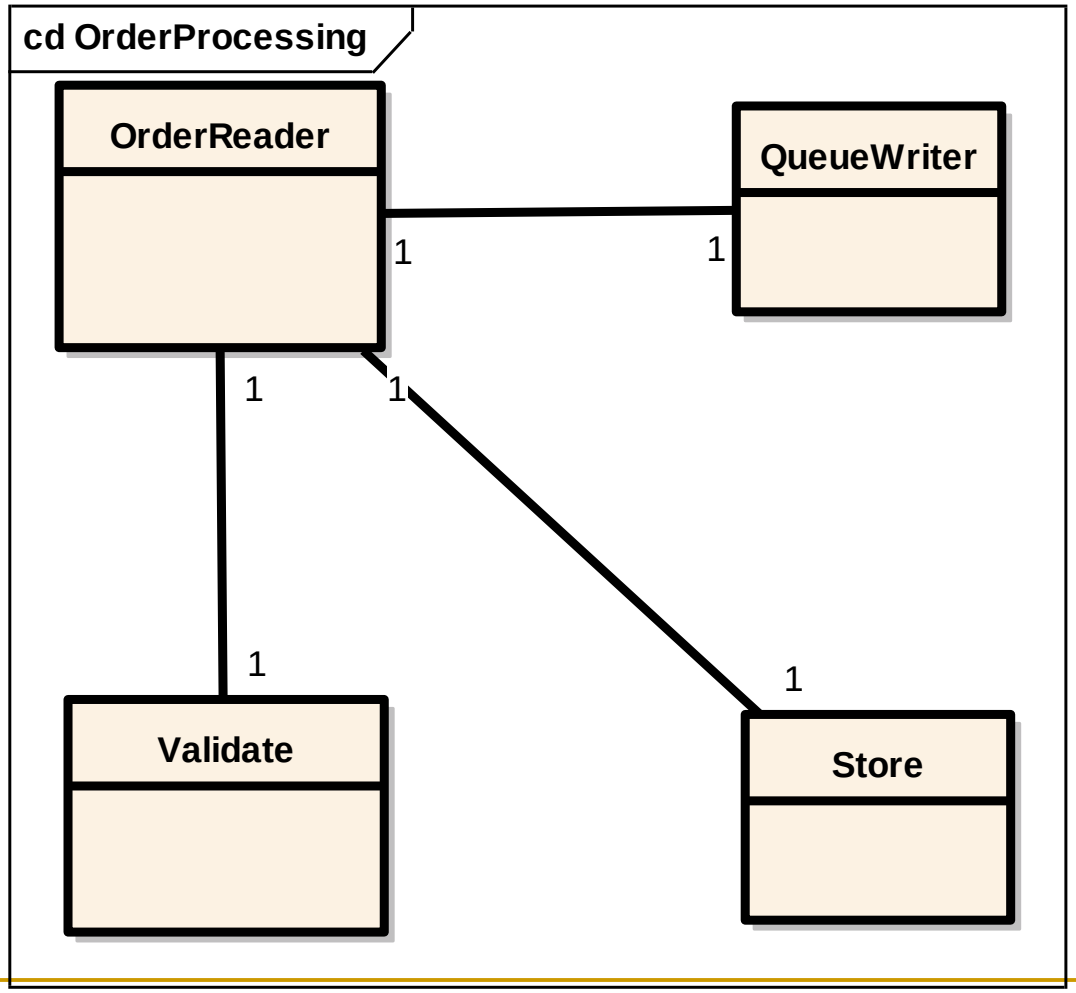
Prototyping Thoughts

- Prototypes should be used judiciously to help reduce the risks inherent in a design.
- Only way to address:
  - Performance
  - Scalability
  - Ease of integration
  - Capabilities of off-the-shelf components
- Need to be carefully scoped and managed.
  - Ideally take a day or two, a week or two at most.
  - Usually thrown-away so keep them cheap
  - Don’t let them acquire a life of their own

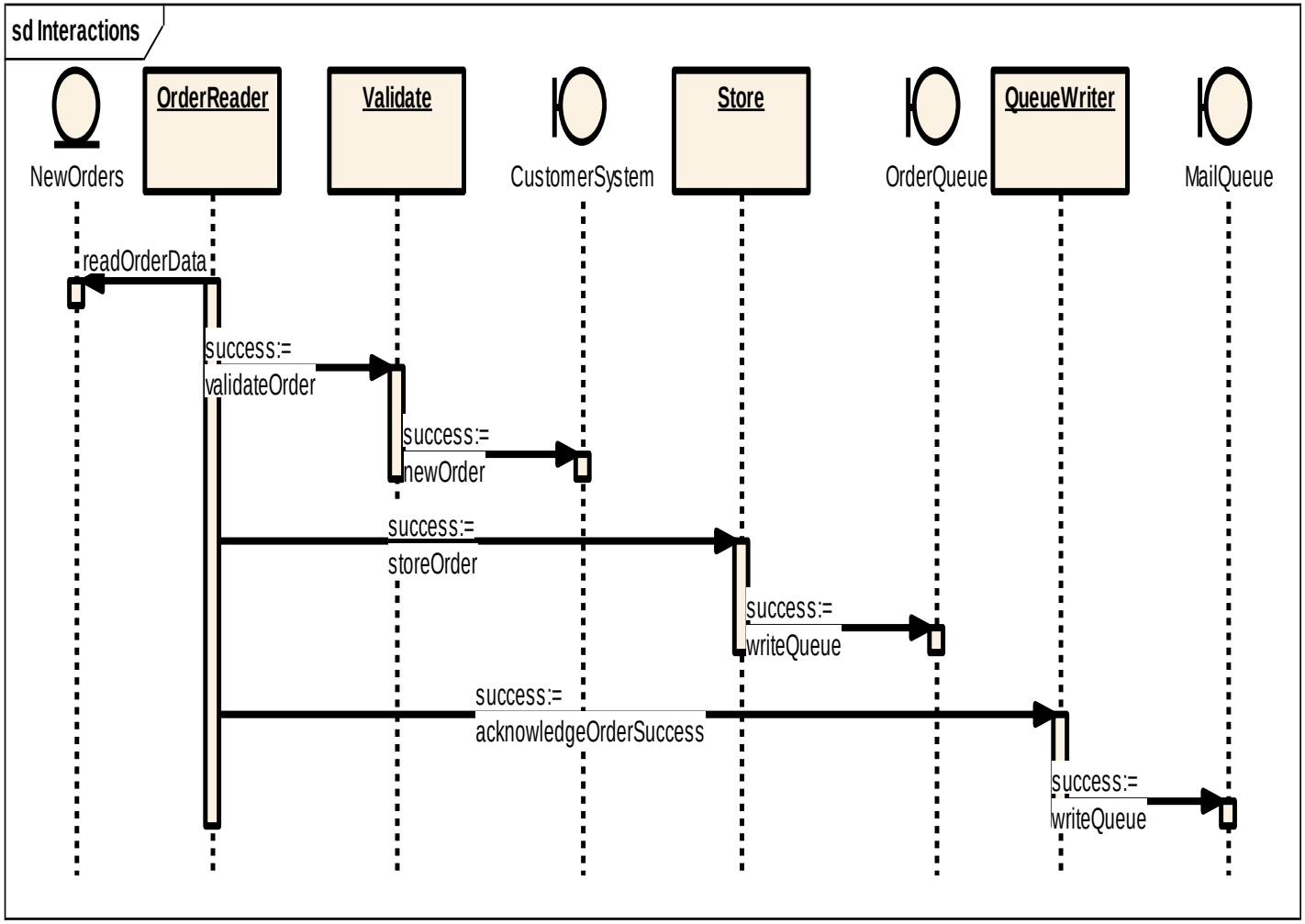
Component Diagram



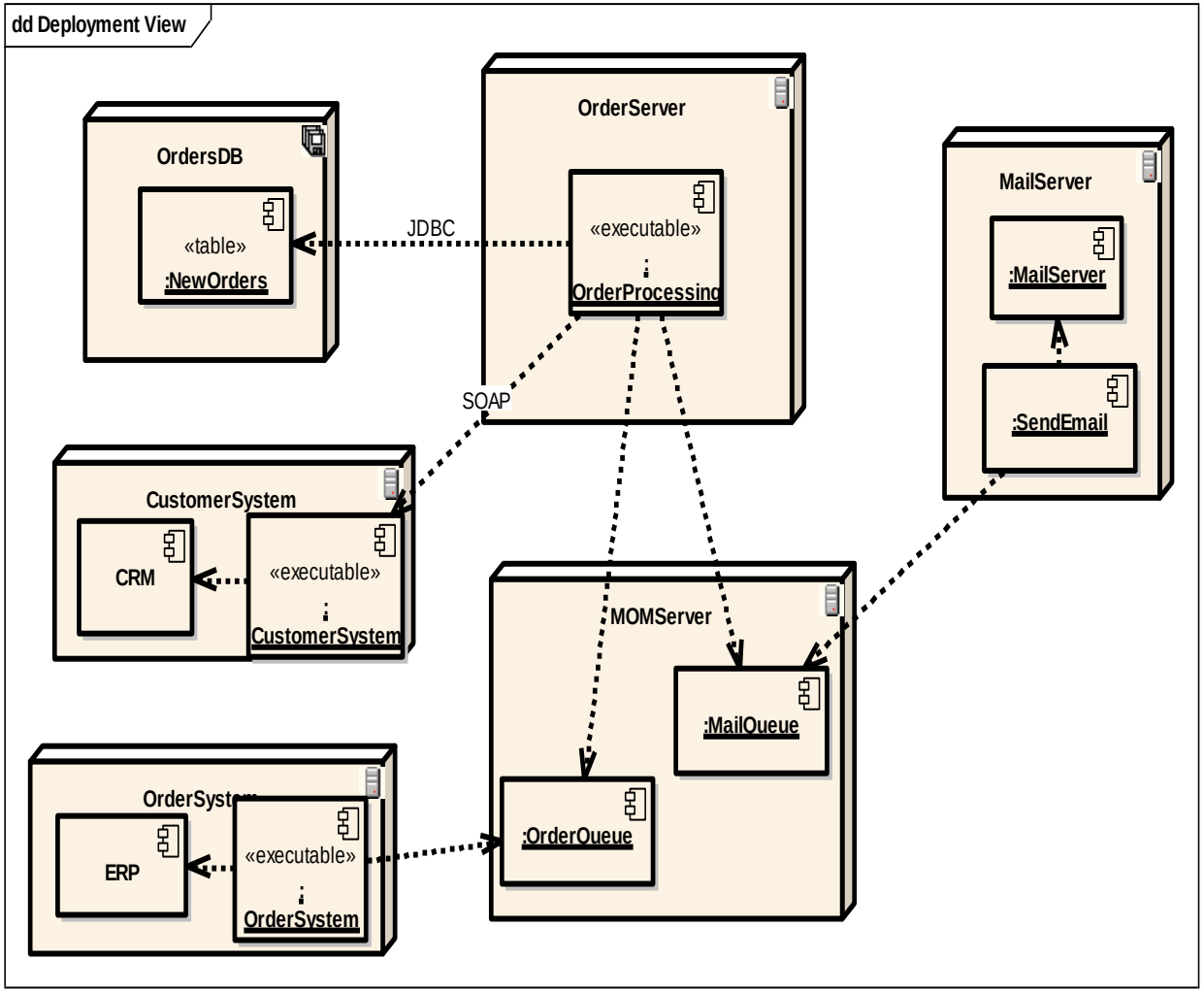
Class Diagram



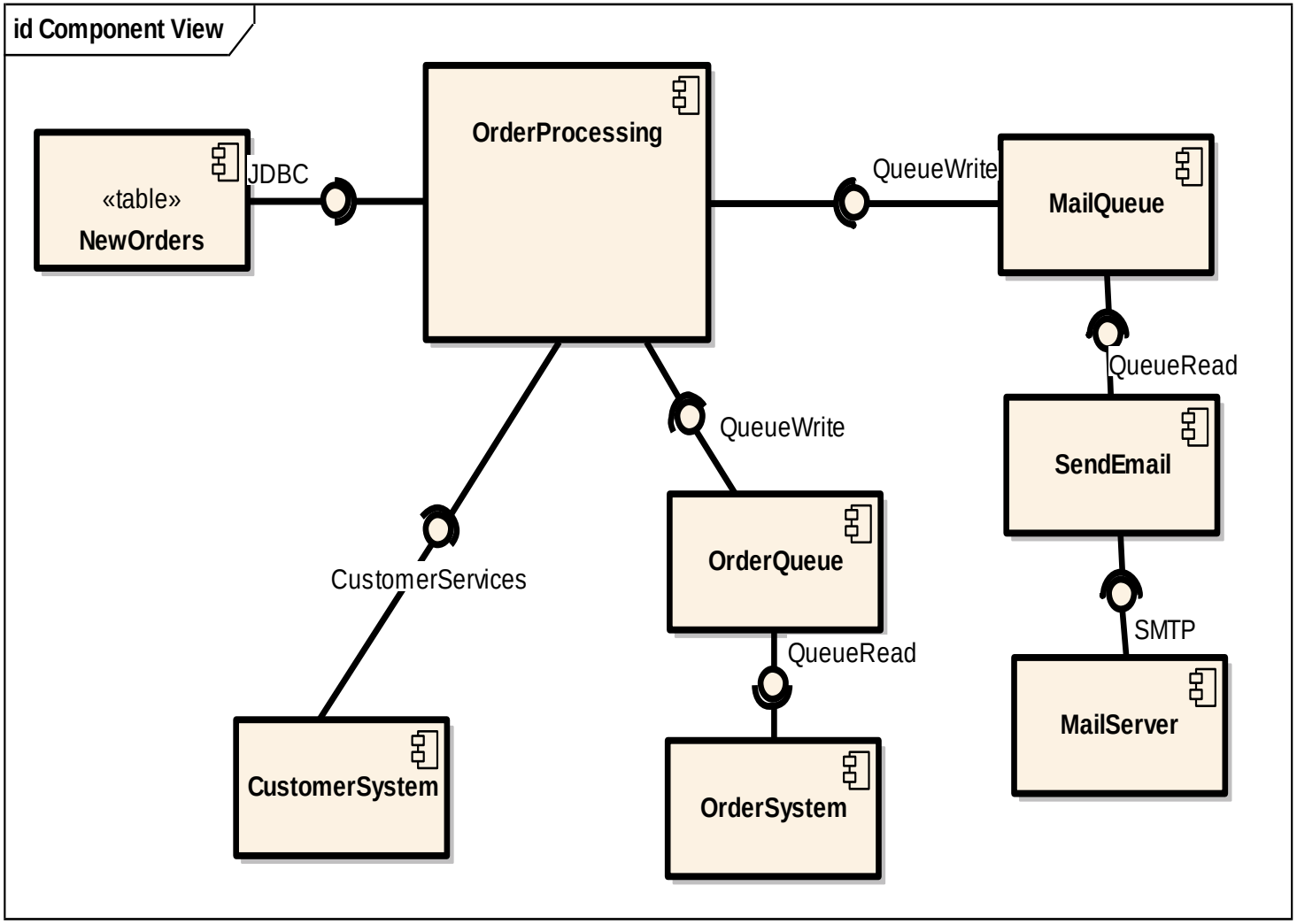
Sequence Diagram



Deployment Diagram



Component Interfaces



Component Decomposition