System Architecture

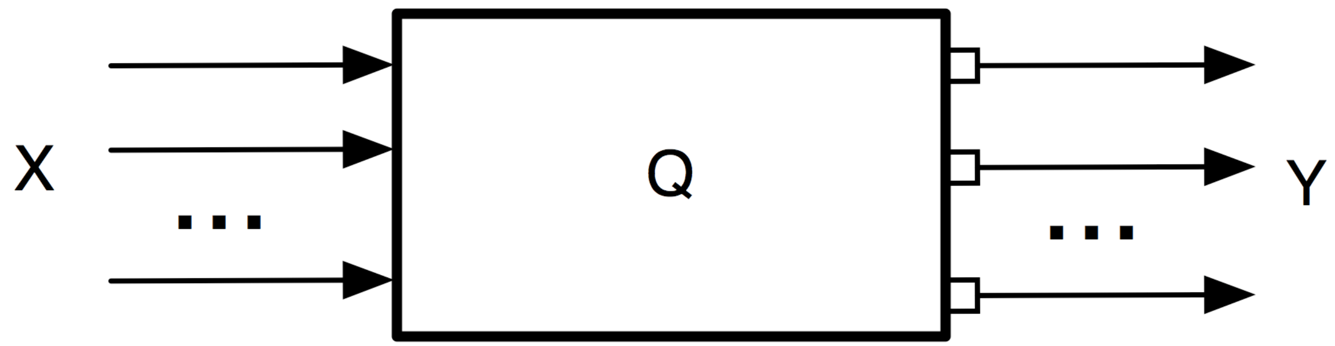
- Principles
  - A Unified Formalism for Complex Systems Architecture
  - Section 1.3
    - http://www.lix.polytechnique.fr/~golden/systems_architecture.html
    - http://www.lix.polytechnique.fr/~golden/research/phd.pdf
- Development
  - Microsoft Application Architecture Guide
    - Chapter 1, 2, 4
    - https://msdn.microsoft.com/en-us/library/ff650706.aspx
- Patterns
  - Microsoft Application Architecture Guide
    - Chapter 3
  - An Introduction to Software Architecture
    - http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf

System

- From wikipedia
  - A system is a set of interacting or interdependent components forming an integrated whole.
  - Delineated by its spatial and temporal boundaries,
  - Surrounded and influenced by its environment
  - Described by its structure and purpose
  - Expressed in its functioning.

Principles

Thinking with a systemic approach

- **the objects of the reality are modeled as systems**
  - i.e. a box performing a function and defined by its perimeter, inputs, outputs and an internal state
- 

Principles

Thinking with a systemic approach

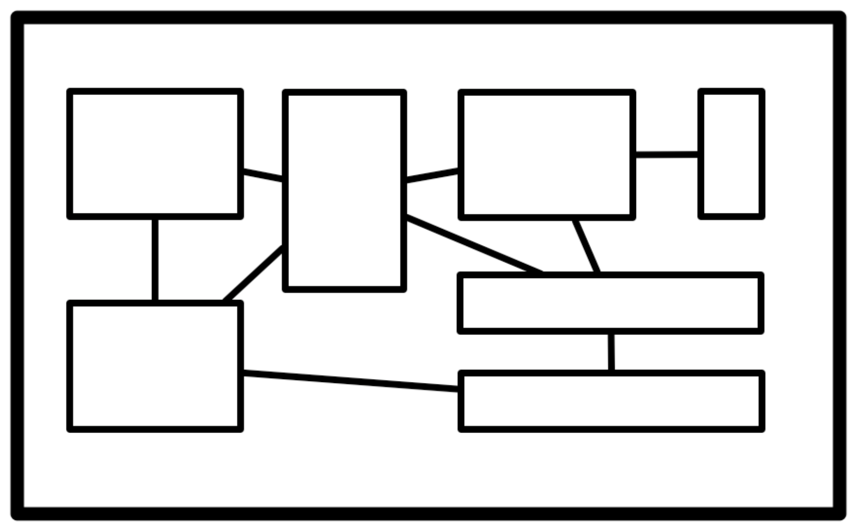
- **the objects of the reality are modeled as systems**

- Ex: a mobile phone is a system which takes in input a voice & keystrokes and outputs voices & displays.
  - Moreover, it can be on, off or in standby.
  - Overall, the phone allows to make phone calls (among other functions).

Principles

Thinking with a systemic approach

- **a system can be broken down into a set of smaller subsystems,**
  - which is less than the whole system
  - Composition emerges new behavious



Principles

Thinking with a systemic approach

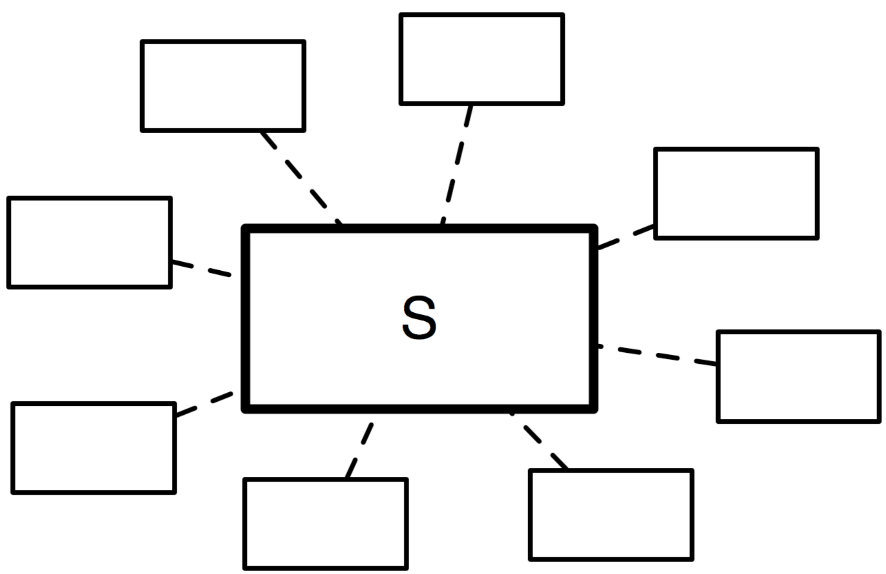
- **a system can be broken down into a set of smaller subsystems,**

- Ex: a mobile phone is in fact a screen, a keyboard, a body, a microphone, a speaker, and electronics.
  - But the phone is the integration of all those elements and cannot be understood completely from this set of elements.

Principles

Thinking with a systemic approach

- **a system must be considered in interaction with other systems**,
  - i.e. its environment
  - Other systems



Principles

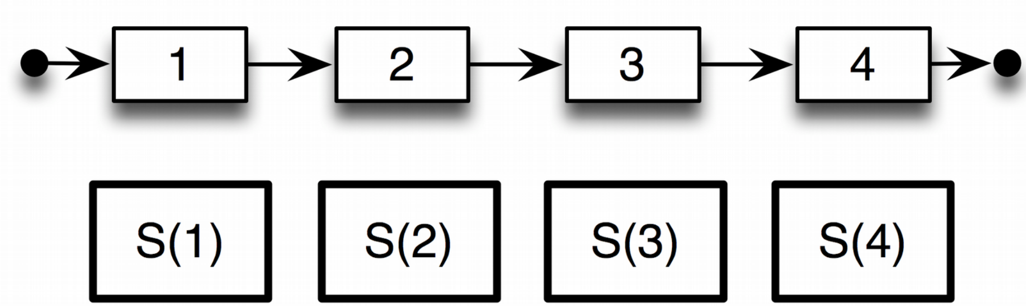
Thinking with a systemic approach

- **a system must be considered in interaction with other systems**,
- Ex: a mobile phone is in interaction with users, antennas (to transmit the signal), reparators (when broken), the ground (when falling), software, etc.
  - All these systems constitute its environment and shall be considered during its design.

Principles

Thinking with a systemic approach

- a system must be considered through its whole lifecycle
  - From the moment it sarts being produced
  - Sold to consummer
  - Switched on
  - Switched off
  - Malfunction...



Principles

Thinking with a systemic approach

- a system must be considered through its whole lifecycle

- Ex: a mobile phone will be designed, prototyped, tested, approved, manufactured, distributed, selled, used, repaired, and finally recycled.
  - All these steps are important (and not only the moment when it is used).

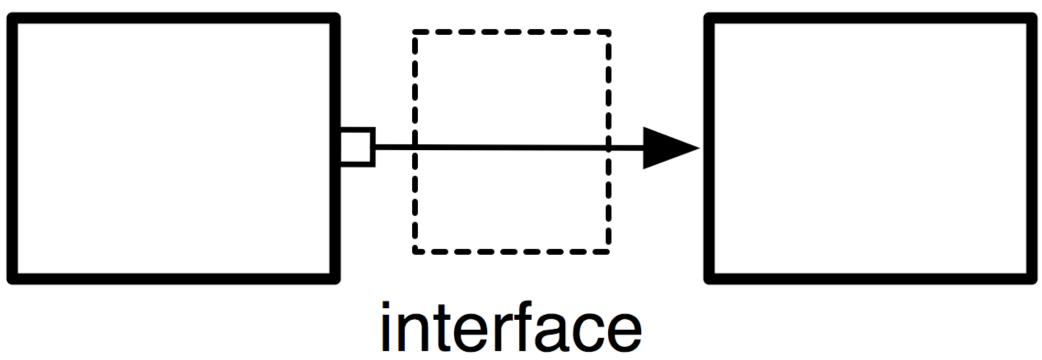
Principles

- Thinking with a systemic approach
  - objects of the reality are modeled as systems
  - system can be broken down into a set of smaller subsystems
  - system must be considered in interaction with other systems
  - system must be considered through its whole lifecycle
  - system can be linked to another through an interface
  - a system can be considered at various abstraction levels
  - system can be viewed according to several layers

Principles

Reasoning according to an architecture paradigm

- **a system can be linked to another through an interface**,
  - which will model the properties of the link
  - The link requirements will model the provided interfaces



Principles

Reasoning according to an architecture paradigm

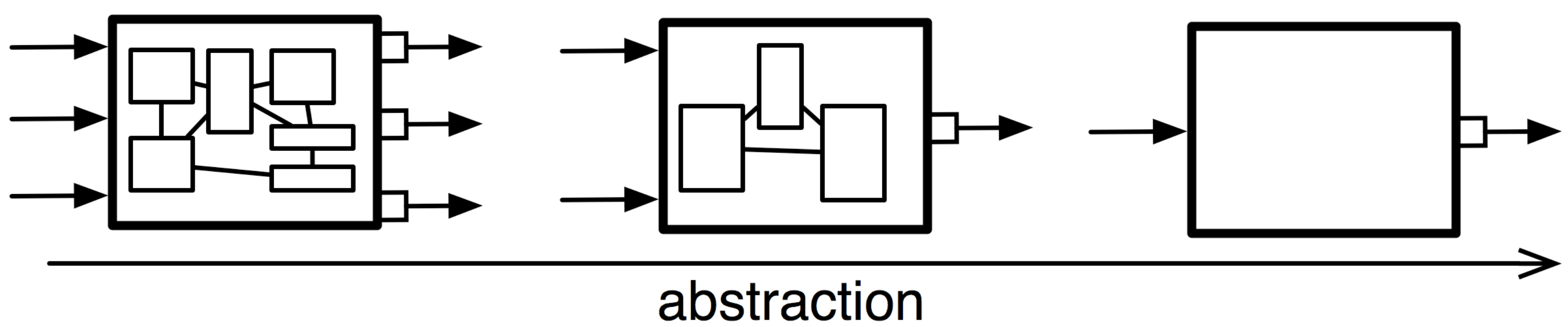
- **a system can be linked to another through an interface,**

- Ex: when phoning, our ear is in direct contact with the phone, and there is therefore a link between the two systems -the ear and the phone
  - However, there is a hidden interface : the air! The properties of the air may influence the link between the ear and the phone (imagine for example if there is a lot of noise).

Principles

Reasoning according to an architecture paradigm

- a system can be considered at various abstraction levels,
  - allowing to consider only relevant properties and behaviors



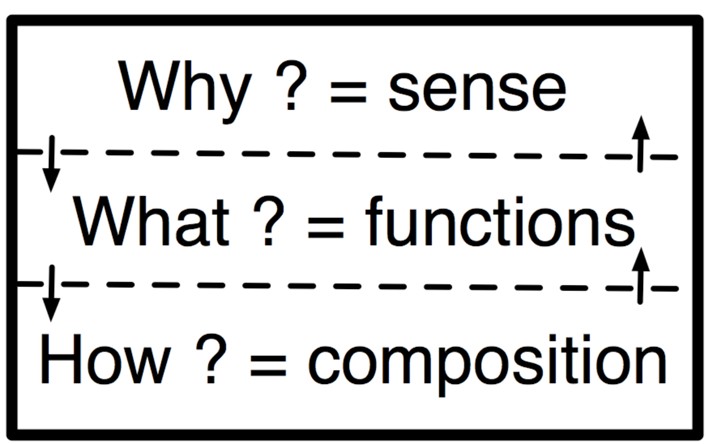
Principles

Reasoning according to an architecture paradigm

- a system can be considered at various abstraction levels,
- Ex: do you consider your phone as a device to make phone calls (and other functions of modern phones), a set of material and electronics components manufactured together, or a huge set of atoms ?
  - All these visions are realistic, but they are just at different abstraction levels, whose relevancy will depend on the context.

Principles

Reasoning according to an architecture paradigm

- a system can be viewed according to several layers
  - its sense – why is it being produced (offered functionality)
  - Functions – operation to fulfill offered functionality
  - Composition – definition of components/subsystems necessary to implement the functions
- 

Principles

Reasoning according to an architecture paradigm

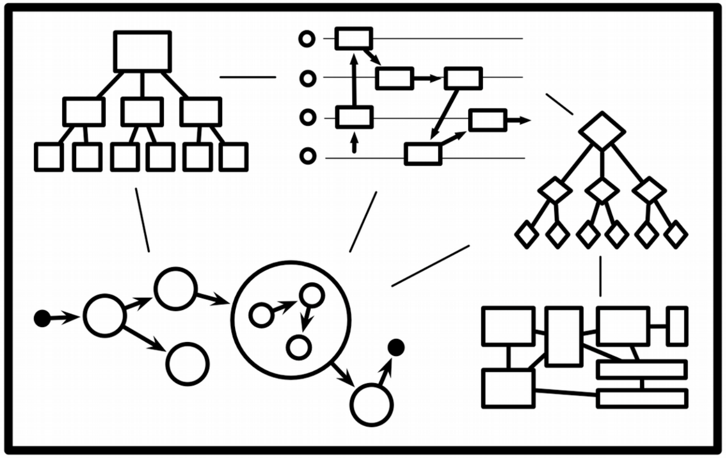
- **a system can be viewed according to several layers**

- A phone is an object whose sense is to accomplish several missions for its environment
  - making phone calls, being a fashionable object, offering various features of personal digital assistants, etc.
- Is a set of functions organized to accomplish these missions
  - displaying on the screen, transmitting signal, delivering power supply, looking for user inputs, making noise if necessary, etc).
- Finally, all these functions are implemented by physical components:
  - Antenna, communication co-processor, network stack

Principles

Reasoning according to an architecture paradigm

- **a system can be described through interrelated models with given semantics**
  - Properties that the system should provide (requirements)
  - Structure of the various components (how they interact)
  - States of the system
  - Behaviors of the system
  - Manipulated data, etc
- This will typically be described with diagrams in SysML (an evolution of UML)



Principles

Reasoning according to an architecture paradigm

- **a system can be described through interrelated models with given semantics**

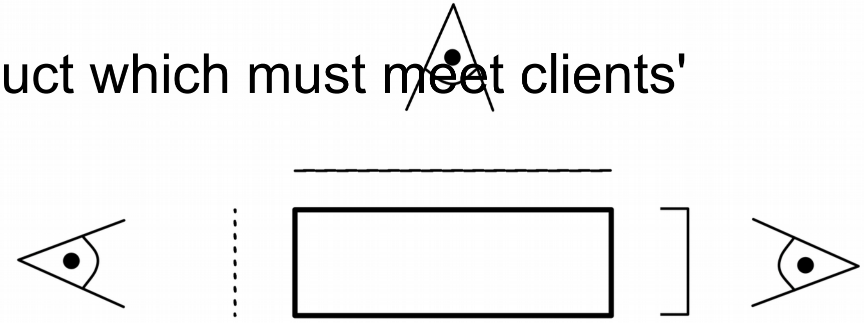
- From the point of view of properties, the phone is a device expected to meet requirements like
  - "a phone must resist to falls from a height of one meter".
- But a phone will also change state :
  - when a phone is off and that the power button is pressed, the phone shall turn on.
- Function dynamics of the phone are also relevant:
  - when receiving a call, the screen will display the name and the speaker will buzz, but if the user presses no button the phone will stop after 30 seconds..

Principles

Reasoning according to an architecture paradigm

- **a system can be described through different viewpoints**
  - corresponding to various actors concerned by the system.
  - All these visions are important and define the system in multiple and complementary ways.

- Ex: commercials, designers, engineers (in charge of software, electronics, acoustics, materials, etc) users, repairers...
- All these people will have different visions of the phone.
  - When the designer will see the phone as an easy-to-use object centered on the user, the engineer will see it as a technological device which has to be efficient and robust.
  - A commercial may rather see it as a product which must meet clients' needs and market trends to be sold.



Principles

- Reasoning according to an architecture paradigm
  - system can be linked to another through an interface
  - a system can be considered at various abstraction levels
  - system can be viewed according to several layers
  - a system can be described through interrelated models with given semantics
  - a system can be described through different viewpoints

Systems Architecture

- The architecture of a system
  - i.e. a model to describe/analyze a system
- Architecting a system
  - i.e. a method to build the architecture of a system
- Body of knowledge for "architecting" systems
  - while meeting business needs,
  - i.e. a discipline to master systems design.
  - consisting in
    - concepts, principles, frameworks, tools, methods, heuristics, practices

Software Architecture

- Process of defining a structured solution that meets all of the technical and operational requirements,
  - while optimizing common quality attributes such as performance, security, and manageability
- The software architecture of a program or computing system
  - is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.
- Architecture is concerned with the public side of interfaces;
  - private details of elements—details having to do solely with internal implementation—are not architectural

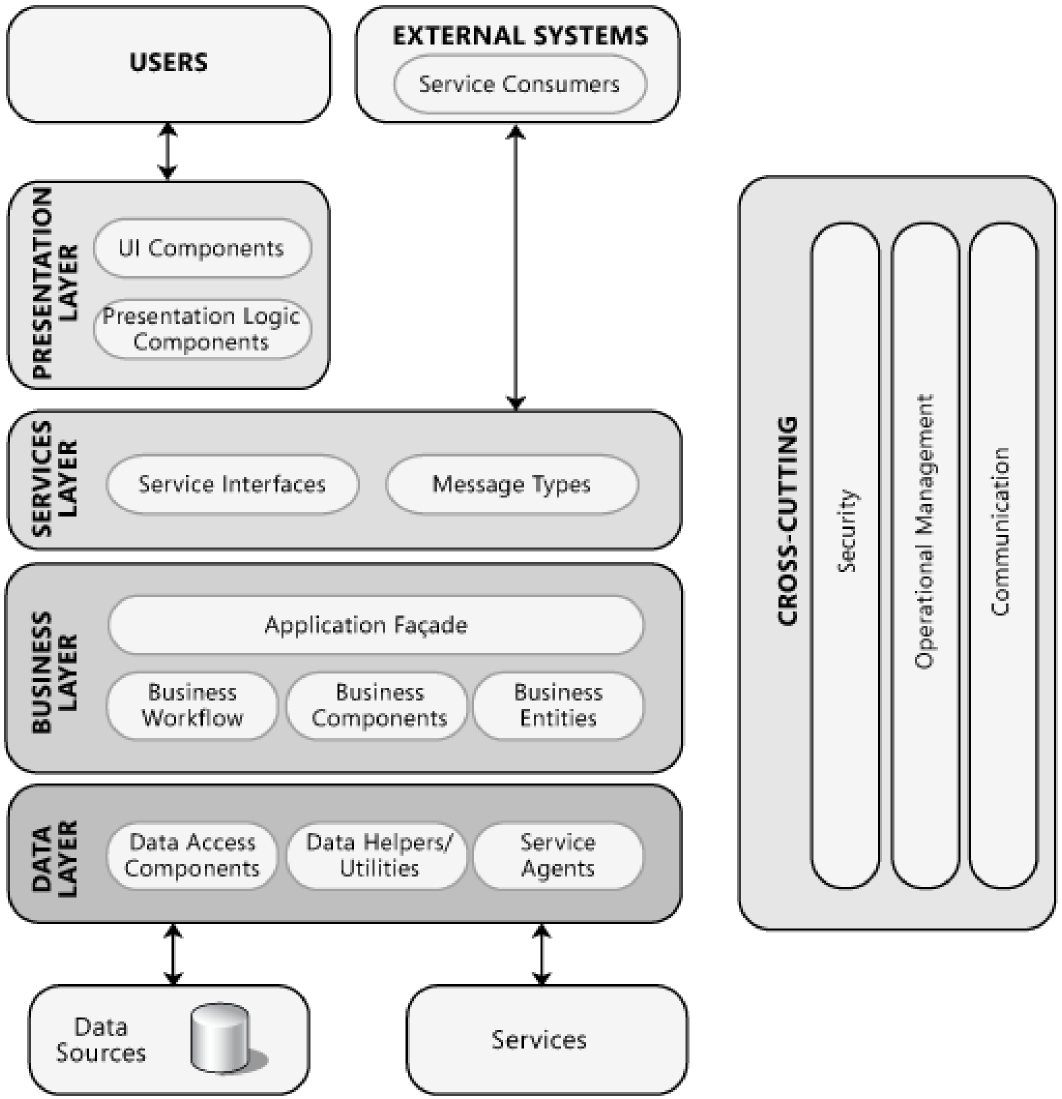
Key Architecture Principles

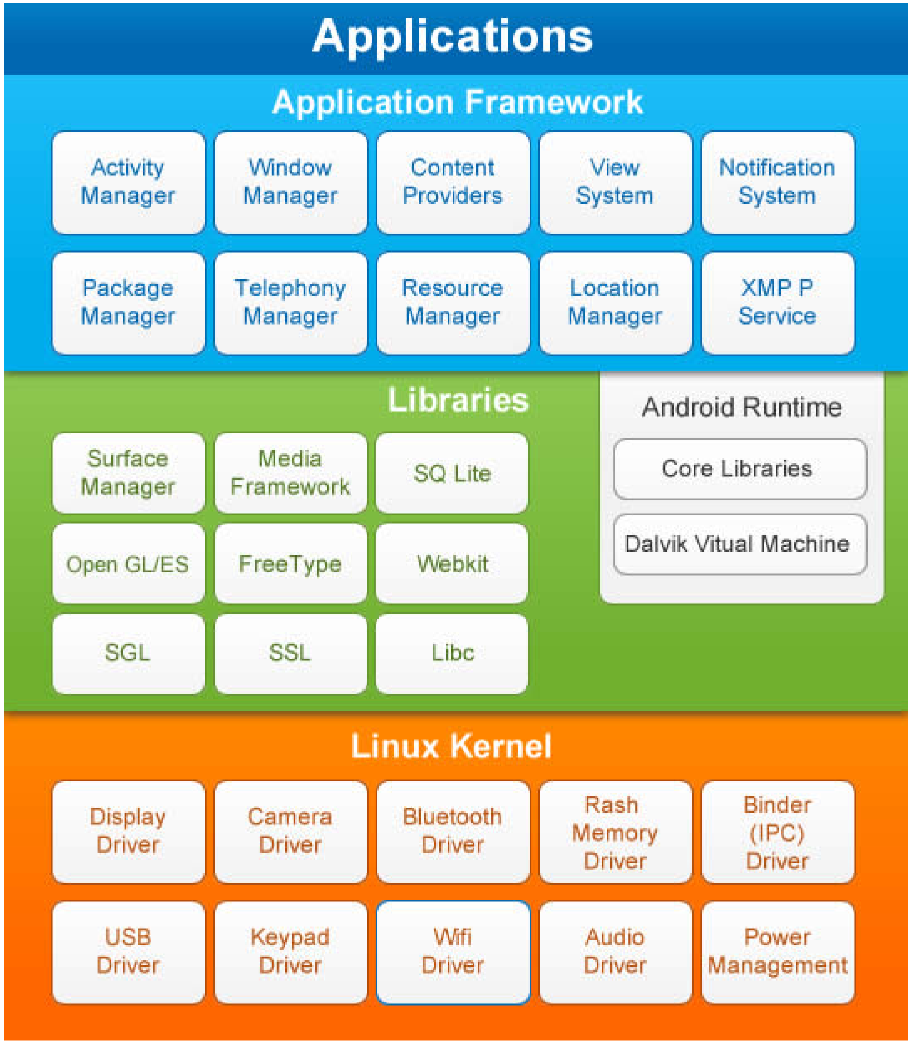
- Build to change instead of building to last
- Model to analyze and reduce risk
- Use models and visualizations as a communication and collaboration tool
- Identify key engineering decisions
- Architecture testing
  - What assumptions have I made in this architecture?
  - What explicit or implied requirements is this architecture meeting?
  - What are the key risks with this architectural approach?
  - What countermeasures are in place to mitigate key risks?
  - In what ways is this architecture an improvement over the baseline or the last candidate architecture?

Key Architecture Principles

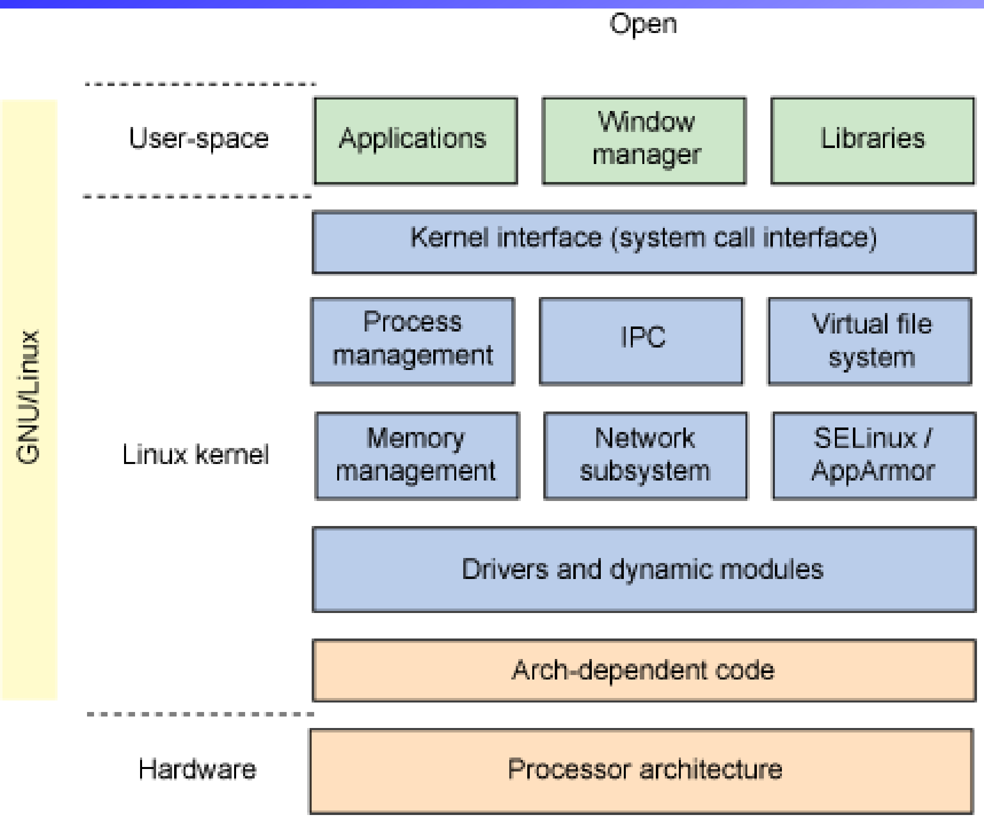
- Common application architecture concerns
  - How to group components
  - How components interact
  - How components work together
  - How components are deployed

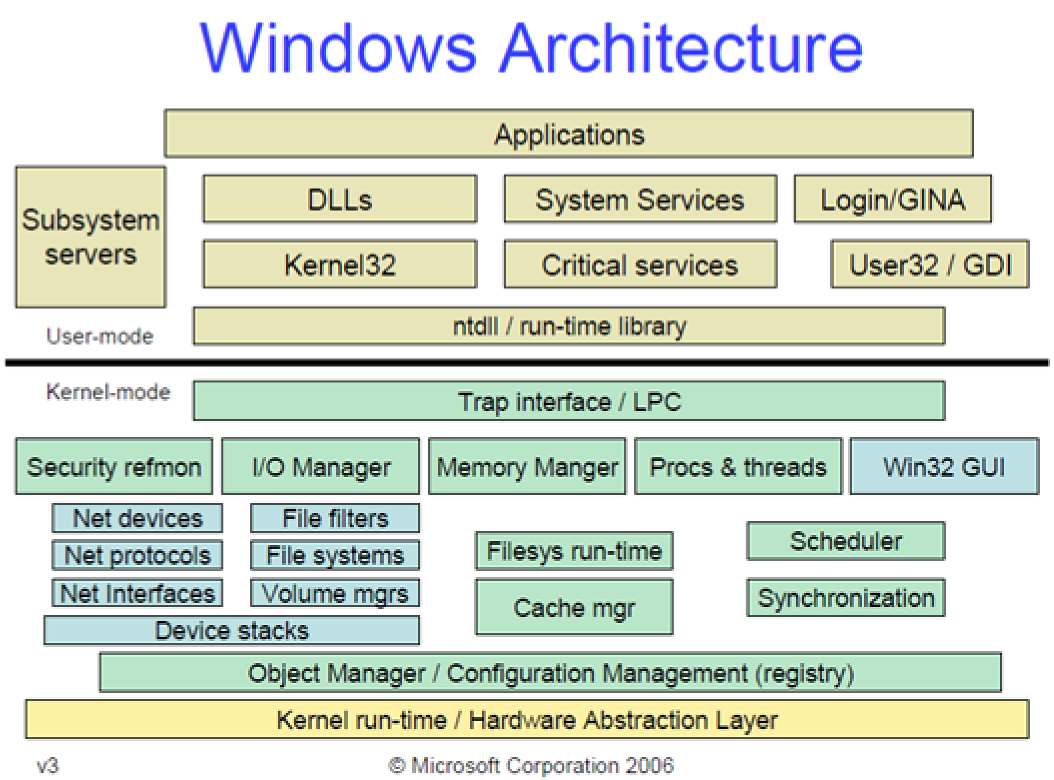
Systems architectures





Systems architectures





Key Design Principles

- Separation of concerns.
  - Divide your application into distinct features with as little overlap in functionality as possible. The important factor is minimization of interaction points to achieve high cohesion and low coupling.
- Single Responsibility principle
  - Each component or module should be responsible for only a specific feature or functionality, or aggregation of cohesive functionality.
- Don’t repeat yourself (DRY).
  - You should only need to specify intent in one place.
  - For example, specific functionality should be implemented in only one component; the functionality should not be duplicated in any other component.

Key Design Principles

- Principle of Least Knowledge (also known as the Law of Demeter or LoD).
  - A component or object should not know about internal details of other components or objects.
- Minimize upfront design.
  - Only design what is necessary. In some cases, you may require upfront comprehensive design and testing if the cost of development or a failure in the design is very high.
  - In other cases, especially for agile development, you can avoid big design upfront.

Design Practices

- Keep design patterns consistent within each layer.
  - Within a logical layer, where possible, the design of components should be consistent for a particular operation
  - However, you may need to use different patterns for tasks in a layer that have a large variation in requirements, such as an application that contains business transaction and reporting functionality.
- Do not duplicate functionality within an application.
  - There should be only one component providing a specific functionality—this functionality should not be duplicated in any other component.
  - This makes your components cohesive and makes it easier to optimize the components if a specific feature or functionality changes.
- Prefer composition to inheritance.
  - Inheritance increases the dependency between parent and child classes, thereby limiting the reuse of child classes.
  - This also reduces the inheritance hierarchies, which can become very difficult to deal with.
  - Composition only relies on the interface

Design Practices

- Establish a coding style and naming convention for development.
  - This provides a consistent model that makes it easier for team members to review code they did not write
  - leads to better maintainability.
- Maintain system quality using automated Quality Assurance techniques during development.
  - Use unit testing and other automated Quality Analysis techniques, such as dependency analysis and static code analysis, during development.
  - Define behavioral and performance metrics for components and sub-systems
- Consider the operation of your application.
  - Determine what metrics and operational data are required by the IT infrastructure to ensure the efficient deployment and operation of your application.
  - Design your application's components with a clear understanding of their individual operational requirements
  - Use automated QA tools during development to ensure that the correct operational data is provided by your application's components and sub-systems.

Problem

- How to design components?
- How to structure a logical solution?
- How to organize the codebase?

Architectural Patterns and Styles

- An architectural style, sometimes called an architectural pattern
  - is a set of principles—a coarse grained pattern that provides an abstract framework for a family of systems.
  - An architectural style improves partitioning and promotes design reuse by providing solutions to frequently recurring problems.
  - You can think of architecture styles and patterns as sets of principles that shape an application.

Architectural Patterns and Styles

- More specifically, an architectural style determines the
  - vocabulary of components and connectors that can be used in instances of that style,
  - together with a set of constraints on how they can be combined.
    - These can include topological constraints on architectural descriptions (e.g., no cycles).
    - Other constraints—say, having to do with execution semantics—might also be part of the style definition."

Architectural Patterns and Styles

Category	**Architecture styles**
Structure	Component-Based Object-Oriented Data Abstraction Layered Architecture Table Driven Interpreters State transition systems
Deployment	Client/Server N-Tier / 3-Tier Peer-to-peer
Communication	RPC Service-Oriented Architecture (SOA Message Bus, Pipes and Filters, Repositories Event-based, Implicit Invocation

Problem

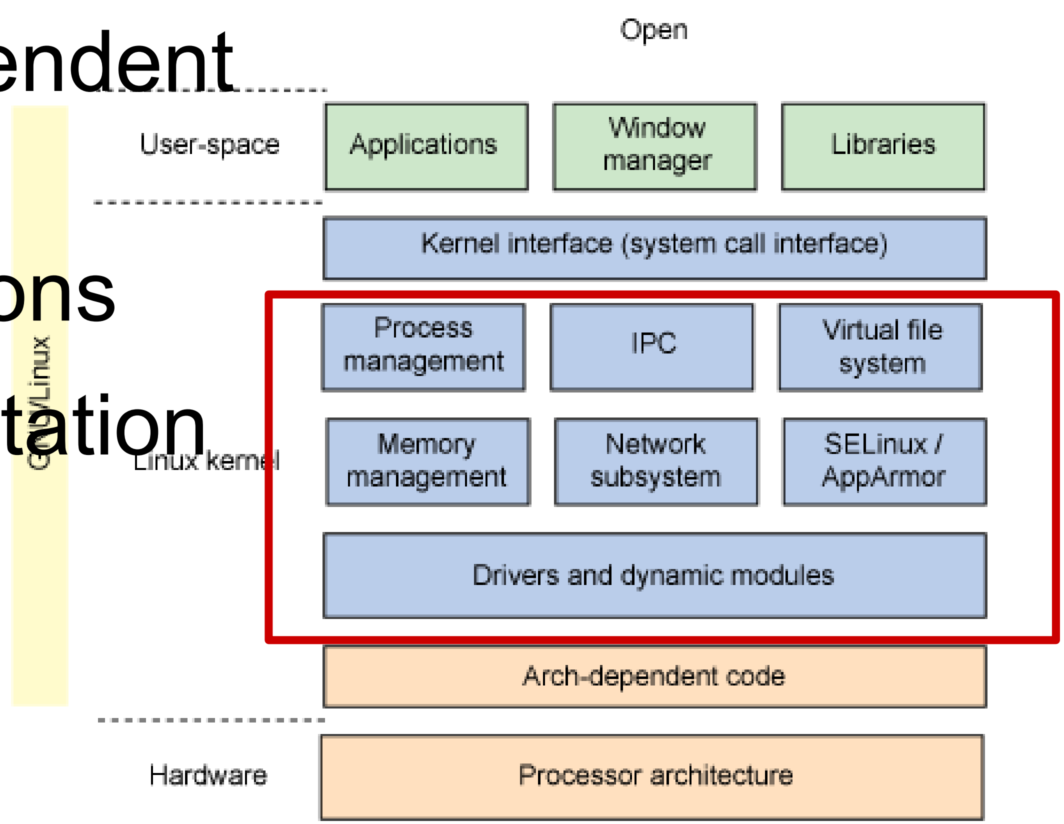
- How to design components?
- How to structure a logical solution?
- How to organize the codebase?
- Structure
  - Component-Based
  - Object-Oriented
  - Data Abstraction
  - Layered Architecture
  - Table Driven Interpreters
  - State transition systems

Data Abstraction

- data representations and their associated primitive operations are encapsulated in an abstract data type
- The components are modules
  - and structures
- Modules interact through functions and procedures
- Modules should preserve
  - integrity of representation (interface)
  - hide implementation

Data Abstraction

- Each modules is implemented in independent files
- Exports a set of functions
- Hide internal representation



Components, Modules, and Functions

- A component or an object should not rely on internal details of other components or objects.
  - Each component or object should call a method of another object or component, and that method should have information about how to process the request and, if appropriate, how to route it to appropriate subcomponents or other components.
  - Helps to create an application that is more maintainable and adaptable.
- Do not overload the functionality of a component.
  - Overloaded components often have many functions and properties providing business functionality mixed with crosscutting functionality such as logging and exception handling. The result is a design that is very error prone and difficult to maintain.
  - Applying the single responsibility and separation of concerns principles will help you to avoid this.
- Understand how components will communicate with each other.
  - Requires an understanding of the deployment scenarios your application must support.
  - You must determine if all components will run within the same process, or if communication across physical or process boundaries must be supported—perhaps by implementing message-based interfaces.

Components, Modules, and Functions

- Keep crosscutting code abstracted from the application business logic as far as possible.
  - Crosscutting code refers to code related to security, communications, or operational management such as logging and instrumentation.
  - Mixing the code that implements these functions with the business logic can lead to a design that is difficult to extend and maintain. Changes to the crosscutting code require touching all of the business logic code that is mixed with the crosscutting code.
  - Consider using frameworks and techniques (such as aspect oriented programming) that can help to manage crosscutting concerns.
- Define a clear contract for components.
  - Components, modules, and functions should define a contract or interface specification that describes their usage and behavior clearly.
  - The contract should describe how other components can access the internal functionality of the component, module, or function; and the behavior of that functionality in terms of pre-conditions, post-conditions, side effects, exceptions, performance characteristics, and other factors.

Object-Oriented

- Components are objects
  - encapsulate data and associated operations
- Connectors are messages and method invocations
- Advantages
  - “Infinite malleability” of object internals as long as interface is maintained
  - System decomposition into sets of interacting agents
  - Easily produce concurrent systems
- Disadvantages
  - Objects must know identities of other objects
  - Side effects in object method invocations

Component-Based

- Decomposition of the design into individual functional or logical components
  - that expose well-defined communication interfaces containing methods, events, and properties.
- This provides a higher level of abstraction than object-oriented design principles
  - does not focus on issues such as communication protocols and shared state.
- Depend upon a mechanism within the platform that provides an environment in which they can execute,
  - often referred to as component architecture
  - component object model (COM), distributed component object model (DCOM) in Windows,Common Object Request Broker Architecture (CORBA) and Enterprise JavaBeans (EJB)

Component-Based

- The key principle is the use of components that are:
- Reusable.
  - Components are usually designed to be reused in different scenarios in different applications.
- Replaceable.
  - Components may be readily substituted with other similar components.
- Not context specific
  - Specific information, such as state data, should be passed to the component instead of being included in or accessed by the component.
- Extensible
  - A component can be extended from existing components to provide new behavior.
- Encapsulated.
  - Components expose interfaces and do not reveal details of the internal processes or stat
- Independent
  - Components are designed to have minimal dependencies on other components.

Component-Based

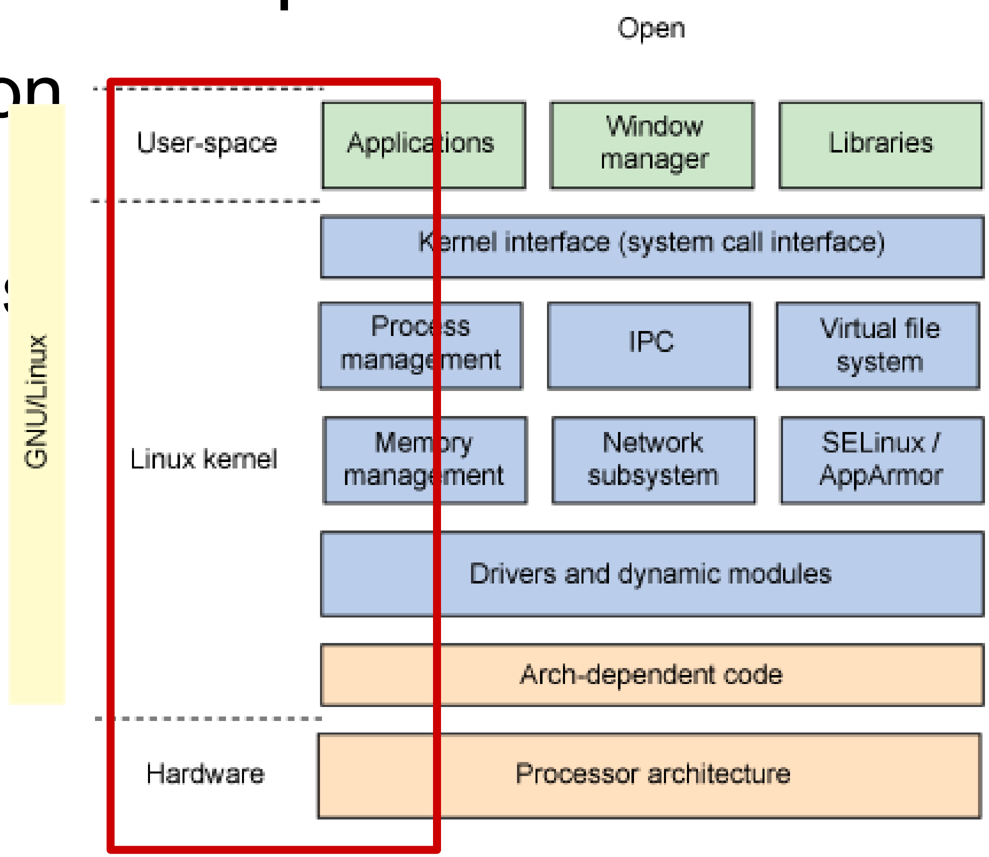
- Ease of deployment
  - As new compatible versions become available, you can replace existing versions with no impact on the other components or the system as a whole.
- Reduced cost.
  - The use of third-party components allows you to spread the cost of development and maintenance.
- Ease of development.
  - Components implement well-known interfaces to provide defined functionality, allowing development without impacting other parts of the system.
- Reusable.
  - The use of reusable components means that they can be used to spread the development and maintenance cost across several applications or systems.
- Mitigation of technical complexity.
  - Use of a component container and its services.
  - Example component services include component activation, lifetime management, method queuing, eventing, and transactions.

Layered Architecture

- Focuses on the grouping of related functionality within an application into distinct layers.
  - Functionality within each layer is related by a common role or responsibility.
- A layered system is organized hierarchically,
  - each layer providing service to the layer above it and serving as a client to the layer below.
- Communication between layers is explicit and loosely coupled.
- Helps to support a strong separation of concerns that, in turn, supports flexibility and maintainability.

Layered Architecture

- The most widely known examples
  - Layered commu-nication protocols
  - Other application areas for this style include database systems and operating systems



Layered Architecture

- Support design based on increasing levels of abstraction.
  - Allows implementors to partition a complex problem into a sequence of incremental steps.
- Support enhancement.
  - Each layer interacts with at most the layers below and above, changes to the function of one layer affect at most two other layers.
- Support reuse.
  - Like abstract data types, different implementations of the same layer can be used interchangeably, provided they support the same interfaces to their adjacent layers.
-

Layered Architecture

- Isolation
  - Allows you to isolate technology upgrades to individual layers in order to reduce risk and minimize impact on the overall system.
- Manageability
  - Separation of core concerns helps to identify dependencies, and organizes the code into more manageable sections.
- Performance.
  - Distributing the layers over multiple physical tiers can improve scalability, fault tolerance, and performance.
- Testability.
  - Increased testability arises from having well-defined layer interfaces, as well as the ability to switch between different implementations of the layer interfaces.

Layered Architecture

- Not all systems are easily structured in a layered fashion.
  - considerations of performance may require closer coupling between logically high-level functions and their lower-level implementations.
- It can be quite difficult to find the right levels of abstraction.
  -

Application Layers

- Separate the areas of concern.
  - Break your application into distinct features that overlap in functionality as little as possible.
  - A feature or functionality can be optimized independently of other features
  - The fail of one functionality will not cause other features to fail as well, and they can run independently of one another.
  - Makes the application easier to understand and design, and facilitates management of complex interdependent systems.
- Be explicit about how layers communicate with each other.
  - Make explicit decisions about the dependencies between layers and the data flow between them.
- Use abstraction to implement loose coupling between layers.
  - Use Interface types or abstract base classes to define a common interface or shared abstraction (dependency inversion) that must be implemented by interface components.

Application Layers

- Do not mix different types of components in the same logical layer.
  - Start by identifying different areas of concern, and then group components associated with each area of concern into logical layers.
  - For example, the UI layer should not contain business processing components, but instead should contain components used to handle user input and process user requests.
- Keep the data format consistent within a layer or component.
  - Mixing data formats will make the application more difficult to implement, extend, and maintain.
  - Every time you need to convert data from one format to another, you are required to implement translation code to perform the operation and incur a processing overhead.

Table Driven Interpreters

- Virtual machines are used as interpreters
- An interpreter generally has four components:
  - an interpretation engine to do the work,
  - a memory that contains the pseudo-code to be interpreted
  - a representation of the control state of the interpretation engine,
  - a representation of the current state of the program being simulated.
- Interpreters are commonly used to build virtual machines
  - Close the gap between the computing engine expected by program (e.g Java) and the computing engine available in hardware.
  - Java program executed in windows, linux or mac os

Problem

- How to distribute/deploy components sub-systems on different hardware
- how to distribute responsibilities among node?
- Deployment
  - Client/Server
  - N-Tier / 3-Tier
  - Peer-to-peer

Client/Server

- Distributed systems that
  - involve a separate client and server system,
  - and a connecting network.
- The simplest form of client/server system involves a server application that is accessed directly by multiple clients,
  - referred to as a 2-Tier architectural style.
- Describes the relationship between a client and one or more servers,
  - where the client initiates one or more requests (perhaps using a graphical UI),
  - waits for replies,
  - processes the replies on receipt.
- The server typically authorizes the user and then carries out the processing required to generate the result.
- The server may send responses using a range of protocols and data formats to communicate information to the client.

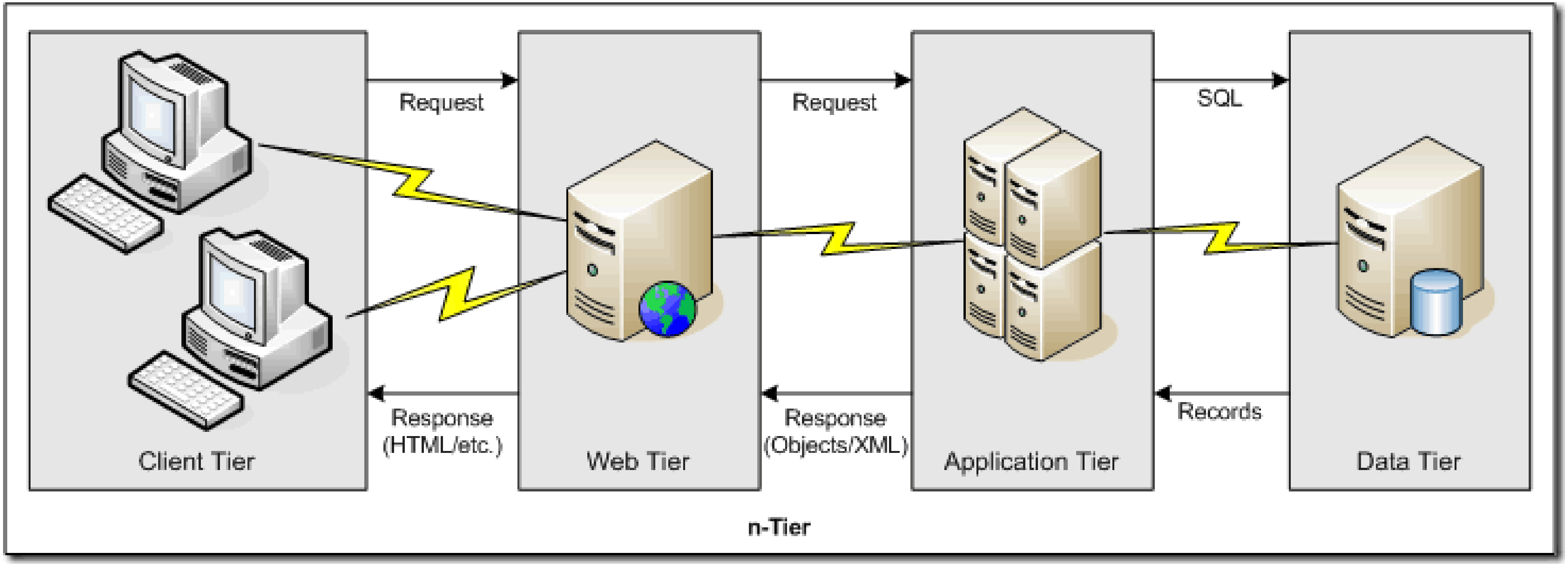
Client/Server

- Higher security.
  - All data is stored on the server, which generally offers a greater control of security than client machines.
- Centralized data access.
  - Because data is stored only on the server, access and updates to the data are far easier to administer than in other architectural styles.
- Ease of maintenance
  - Roles and responsibilities of a computing system are distributed among several servers that are known to each other through a network.
  - Ensures that a client remains unaware and unaffected by a server repair, upgrade, or relocation.

N-Tier / 3-Tier

- Ddescribe the separation of functionality into segments
  - in the same way as the layered style,
  - but with each tier on a separate computer.
- N-tier application architecture is characterized by
  - the functional decomposition of applications, service components
  - their distributed deployment, providing improved scalability, availability, manageability, and resource utilization.
  - Each tier is completely independent from all other tiers, except for those immediately above and below it.

N-Tier / 3-Tier


Request
Request
SQL
Response
(HTML/etc.)
Response
(Objects/XML)
Records
Client Tier
Web Tier
Application Tier
Data Tier
n-Tier

N-Tier / 3-Tier

- Maintainability.
  - Each tier is independent of the other tiers,
  - updates or changes can be carried out without affecting the application as a whole.
- Scalability
  - Tiers are based on the deployment of layers,
  - scaling out an application is reasonably straightforward.
- Flexibility.
  - Each tier can be managed or scaled independently
- Availability
  - Applications can exploit the modular architecture of enabling systems using easily scalable components, which increases availability.

Peer-to-peer

- Client-Queue-Client systems.
  - This approach allows clients to communicate with other clients through a server-based queue.
    - Clients can read data from and send data to a server that acts simply as a queue to store the data.
    - This allows clients to distribute and synchronize files and information.
    - This is sometimes known as a passive queue architecture.

Peer-to-peer

- Peer-to-Peer (P2P) applications.
  - Developed from the Client-Queue-Client style,
  - P2P style allows the client and server to swap their roles in order to distribute and synchronize files and information across multiple clients.
  - It extends the client/ server style through multiple
    - responses to requests,
    - shared data,
    - resource discovery,
    - resilience to removal of peers.

Peer-to-peer

- State and behavior are distributed among peers which can act as either clients or servers.
  - A single component can be a client and a server
- Design ensures users contribute resources to the system
- All nodes have the same capabilities and responsibilities
- Correct operation not dependent on a central system
- Can be design to offer some anonymity
- Data placement (access) algorithm
  - Key issue for efficient operation

Peer-to-peer

- Nodes responsibilities can be replicated
  - Increases Availability
  - Increases reliability,
  - Increases faul-tolerance
- Simple algorithms
  - Eases management, configuration

Problem

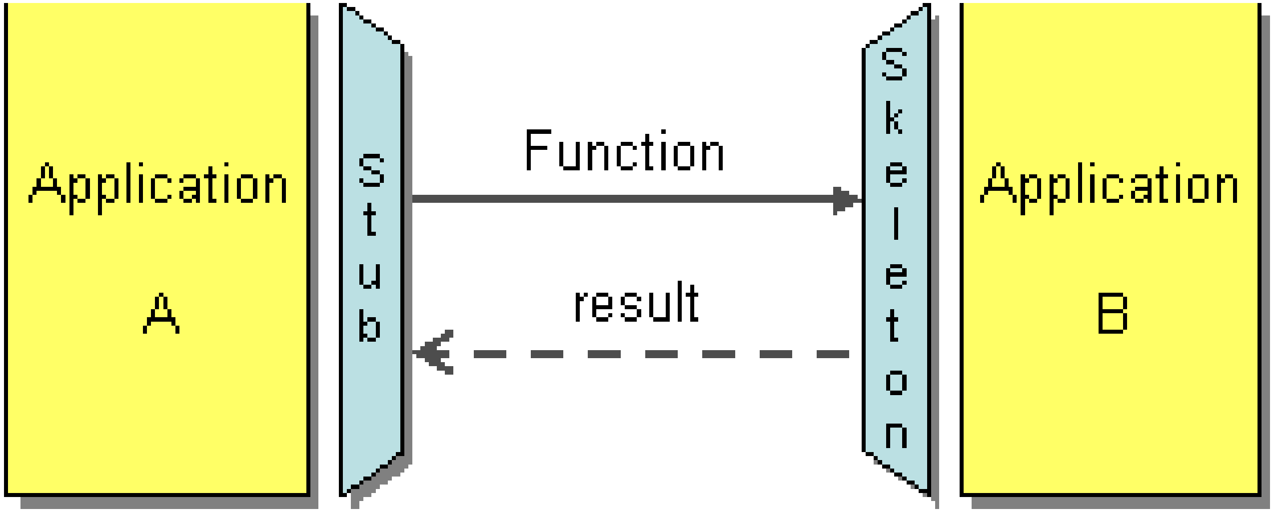
- How to connect 2 components
  - Just one main.c
  - Copy & paste of A.c B.c into main.c
  - gcc main.c
- How to connect 2 components
  - Just one main.c
  - Transform B.c → libB.o A.c → libA.o
  - gcc main.c -lB -lA

Problem

- How to connect 2 components
  - Just one main.c
    - gcc *.c
- How to integrate multiple applications
  - so that they work together
  - and can exchange information
  - and guarantee consistency?
  - With ease of programming
- Communication
  - RPC
  - Service-Oriented Architecture (SOA
  - Message Bus,
  - Pipes and Filters,
  - Repositories
  - Event-based, Implicit Invocation

Problem

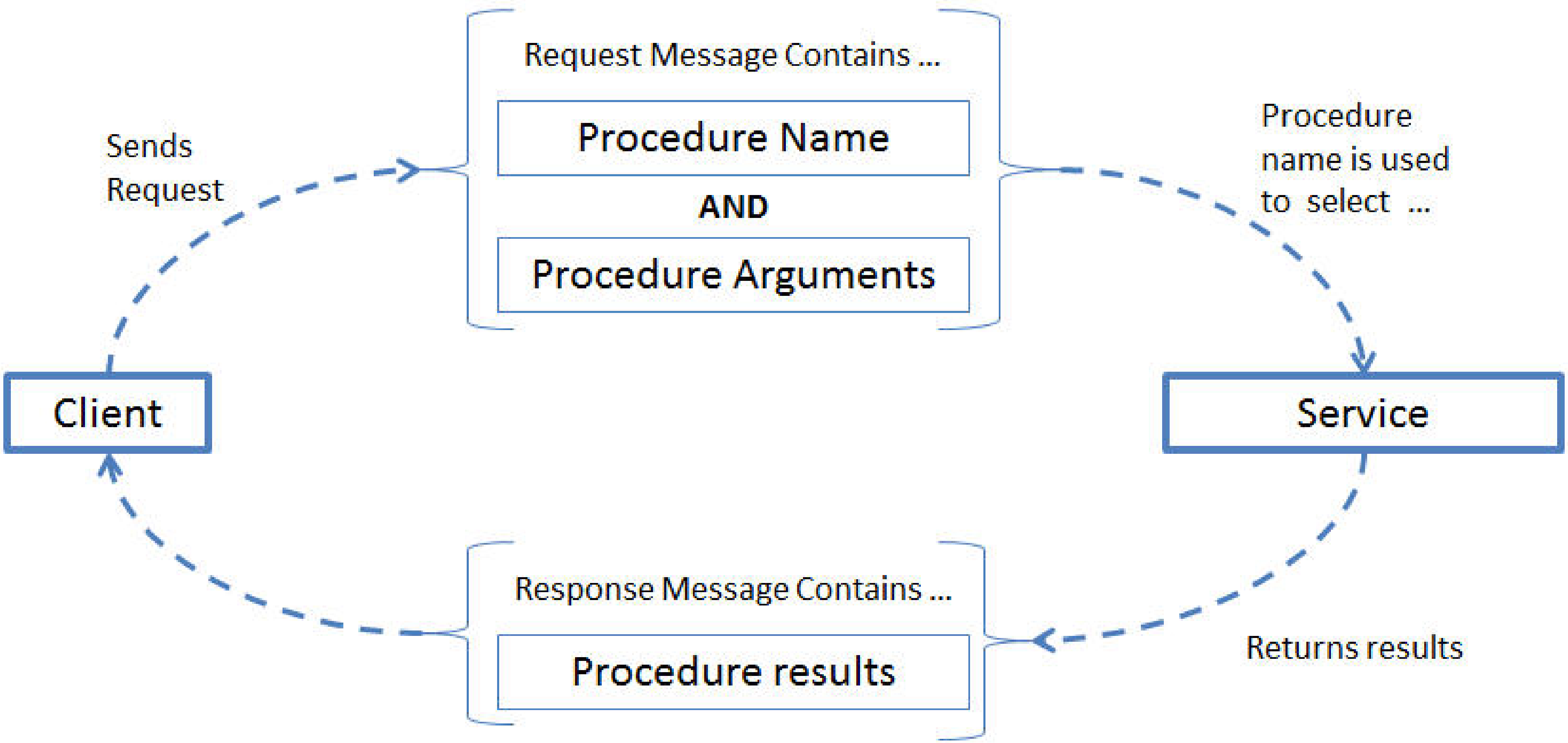
- How to connect 2 components
  - Make them independent components
  - Select a communication pattern
    - RPC
    - Service



Remote Procedure Call

- Component functionality is provided as a set of function calls
  - Applications call methods/function that are executed by a different component
- Is tightly coupled
  - use specific technics to perform communication
  - No standards-based mechanisms to be invoked, published, and discovered
- Application functionality are provided as a set o functions

RPC


Request Message Contains ...
Procedure Name
AND
Procedure Arguments
Sends
Request
Procedure
name is used
to select ...
Client
Service
Response Message Contains ...
Procedure results
Returns results

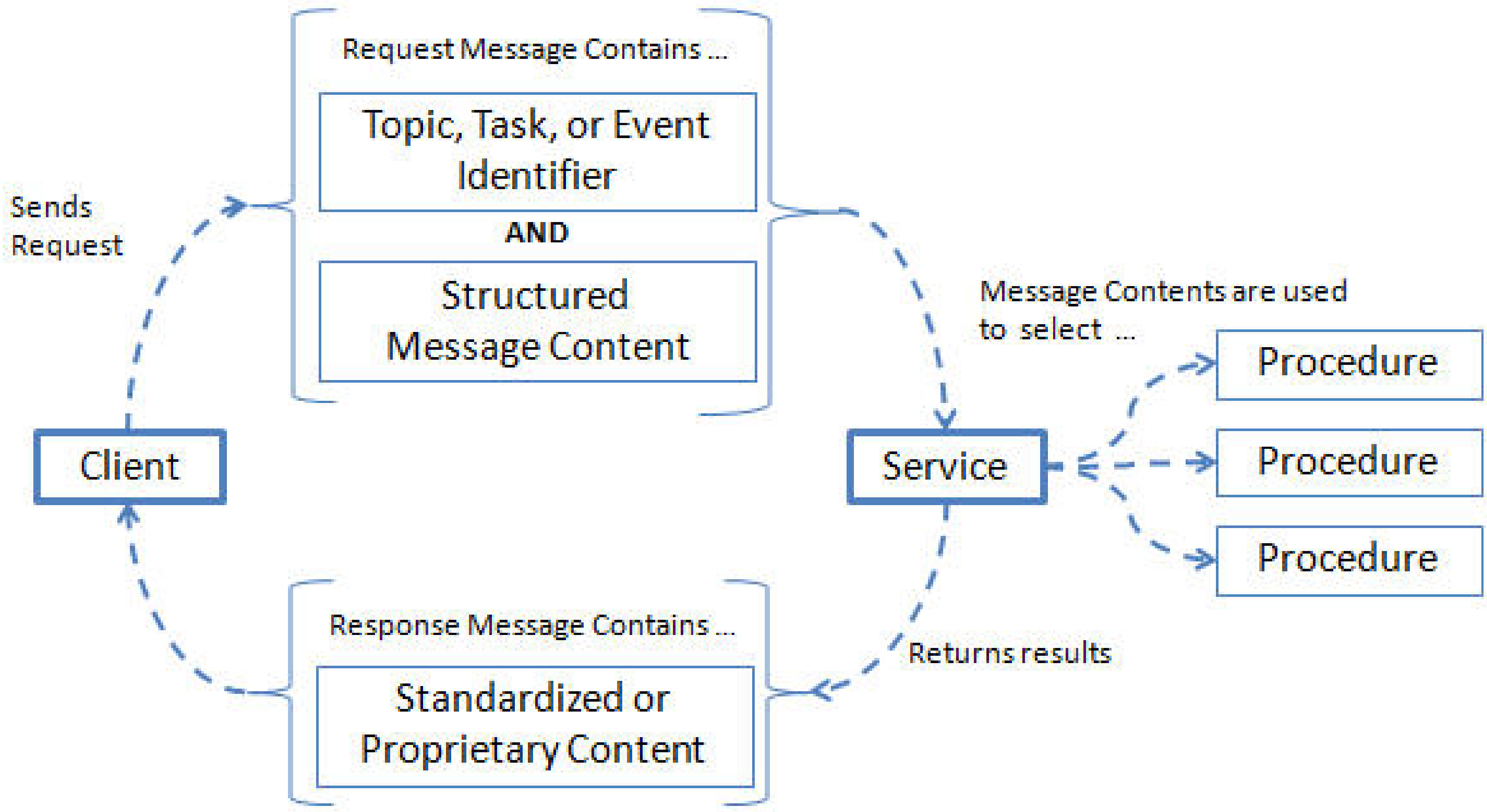
RPC

- Services are autonomous.
  - each service is maintained, developed, deployed, and versioned independently.
- Services are distributable
  - Services can be located anywhere on a network, locally or remotely
  - as long as the network supports the required communication protocols
- Services share interfaces
  - Functions (names, arguments, return)
- Just provides programming abstractions
  - Network implementation hiding

Service-Oriented Architecture

- application functionality is provided as a set of services,
  - Other applications make use of those services
- Services are loosely coupled
  - use standards-based interfaces to be invoked, published, and discovered.

Service-Oriented Architecture



Service-Oriented Architecture

- Services are autonomous.
  - each service is maintained, developed, deployed, and versioned independently.
- Services are distributable
  - Services can be located anywhere on a network, locally or remotely
  - as long as the network supports the required communication protocols.

Service-Oriented Architecture

- Services share schema and contract, not interfaces.
- Compatibility is based on policy
  - Definition of features such as transport, protocol, and security.
- usualy used in component architectures

Service-Oriented Architecture

- Domain alignment
  - Reuse of common services with standard interfaces
  - Increases business and technology opportunities and reduces cost.
- Abstraction
  - Services are autonomous and accessed through a formal contract,
  - provides loose coupling and abstraction.
- **Discoverability**
  - Services can expose descriptions that allow other applications and services to locate them and automatically determine the interface.
- **Interoperability**
  - Formats are based on industry standard
  - provider and consumer of the service can be built and deployed on different platforms
- **Rationalization**
  - Services can be granular in order to provide specific functionality,
  - rather than duplicating the functionality in number of applications, which removes duplication.

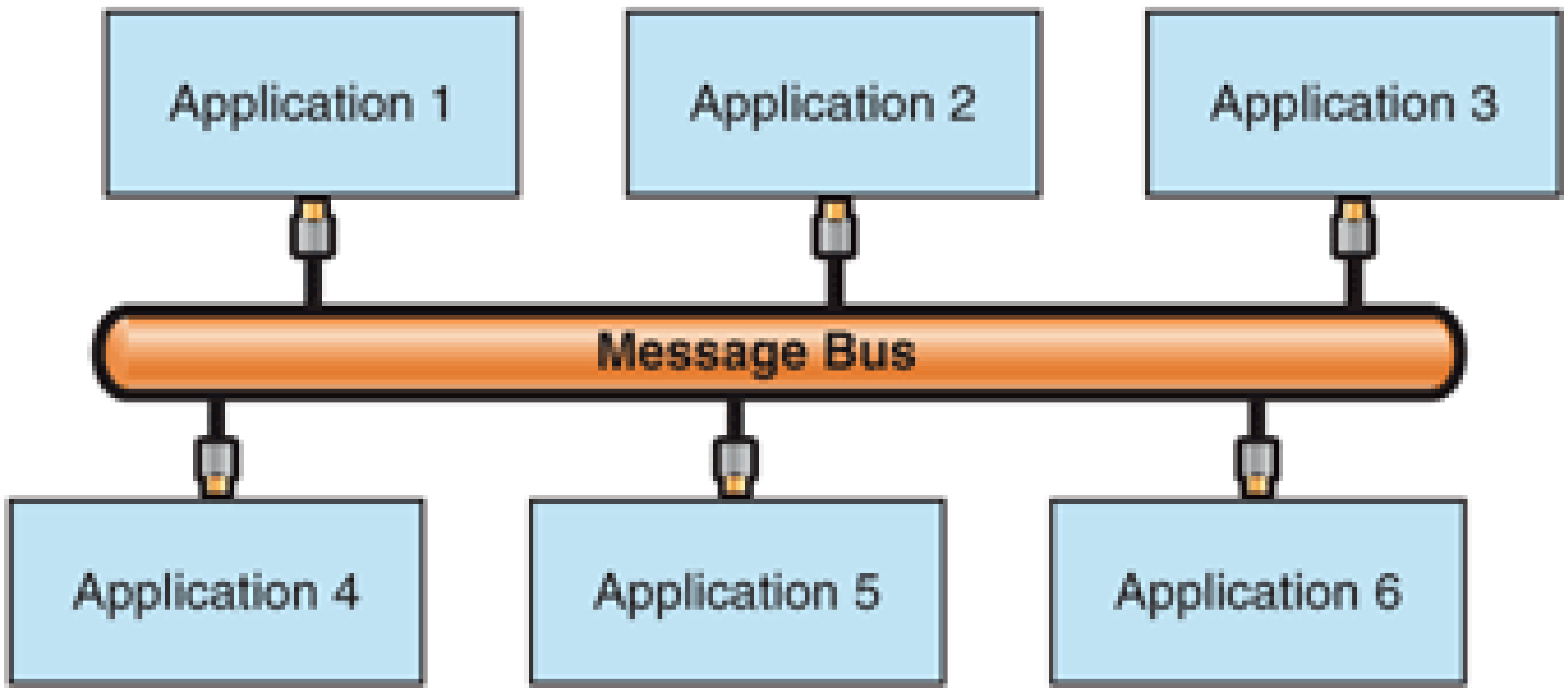
Problem

- How to connect several components
  - Message bus
    - Broadcast
    - Reliability
    - ...
  - Pipes and filters
    - chaining of components

Message Bus

- Describes the principle of using a software system that
  - Can receive and send messages using one or more communication channels
  - without needing to know specific details about each other.
- Interaction between applications is accomplished by passing messages
  - usually asynchronously
  - over a common bus.

Message Bus


Application 1
Application 2
Application 3
Message Bus
Application 4
Application 5
Application 6

Message Bus

- Message-oriented communications.
  - All communication between applications is based on messages
- Complex processing logic.
  - Complex operations can be executed by combining a set of smaller operations
- Modifications to processing logic.
  - interactions based on common schemas and commands,
  - you can insert or remove components on the bus to change the logic that is used to process messages.
- Integration with different environments.
  - Messages follow a common standard
  - Applications (producer and consumers) can be implemented in different technologies

Message Bus

- Extensibility
  - Applications can be added to or removed from the bus
  - without impact on the existing components.
- Low complexity
  - Application only needs to know how to communicate with the bus.
- Flexibility
  - The set of applications that make up a complex process can be changed
  - the communication patterns between applications, can be changed

Message Bus

- Loose coupling
  - with a suitable interface for communication with the message bus,
  - there is no dependency on the application
    - allowing changes, updates, and replacements that expose the same interface.
- Scalability/ fault tolerance
  - Multiple instances of the same application can be attached to the bus in
    - to order to handle multiple requests at the same time.
    - to add fault tolerance

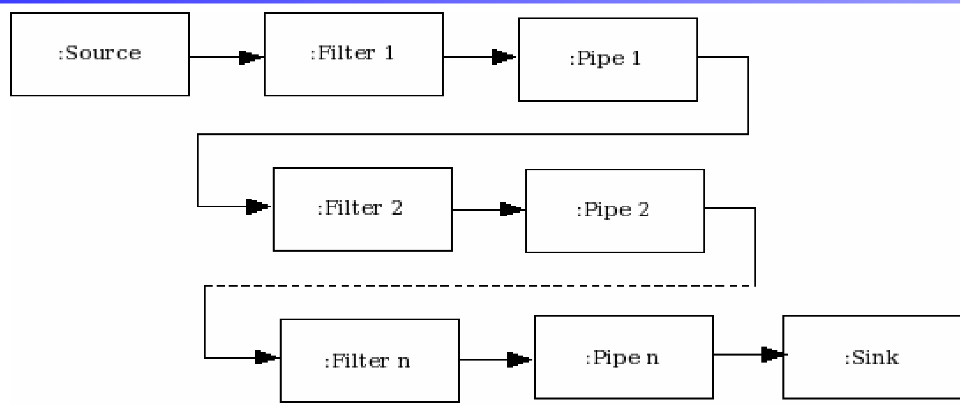
Message Bus

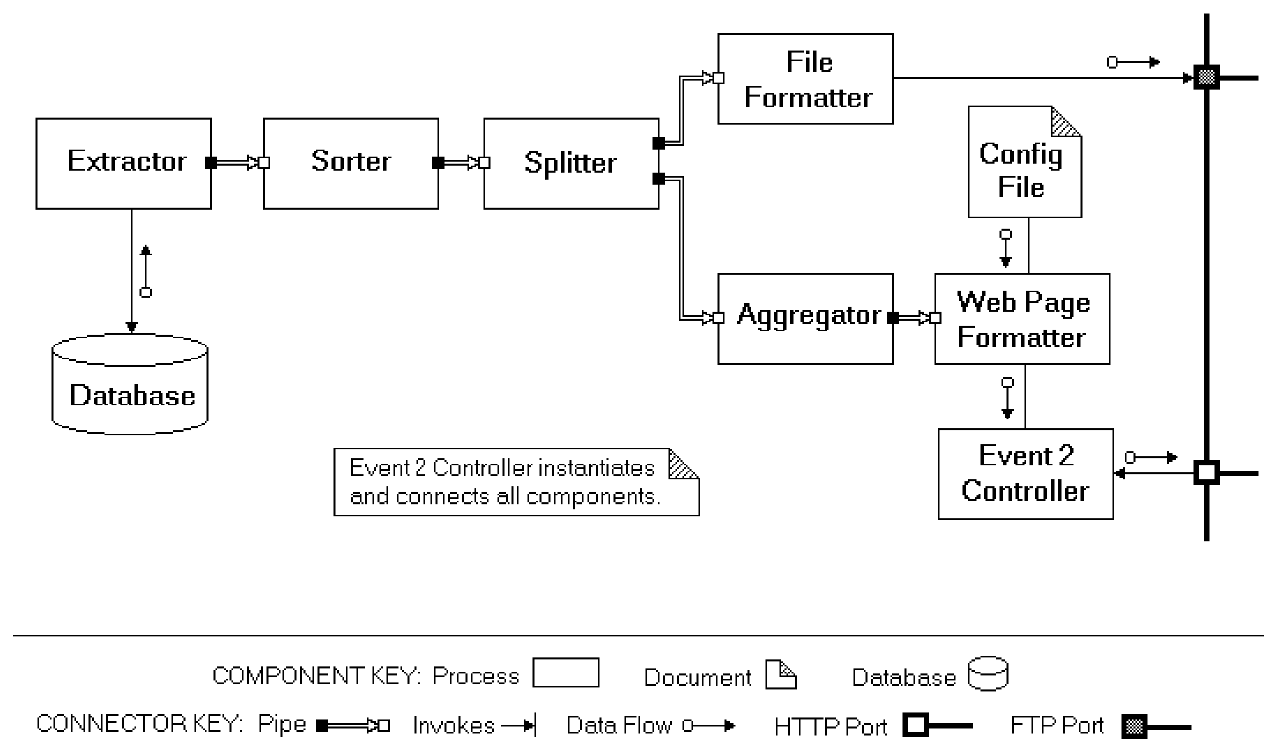
- Complex processing logic
  - Point-to-Point Channel
    - A Point-to-Point Channel ensures that only one receiver consumes any given message.
    - The bus can have multiple receivers
      - but only a single receiver consumes any one message
  - Publish-Subscribe Channel
    - A message is delivered to all consumers that register for that class of messages

Pipes and Filters

- Each component (filter) has a set of inputs and a set of outputs.
- A component reads streams of data on its inputs
  - and produces streams of data on its outputs, delivering a complete instance of the result in a standard order
  - applying a local transformation to the input streams and computing incrementally so output begins before input is consumed.
    - “filters”

Pipes and Filters





Pipes and Filters

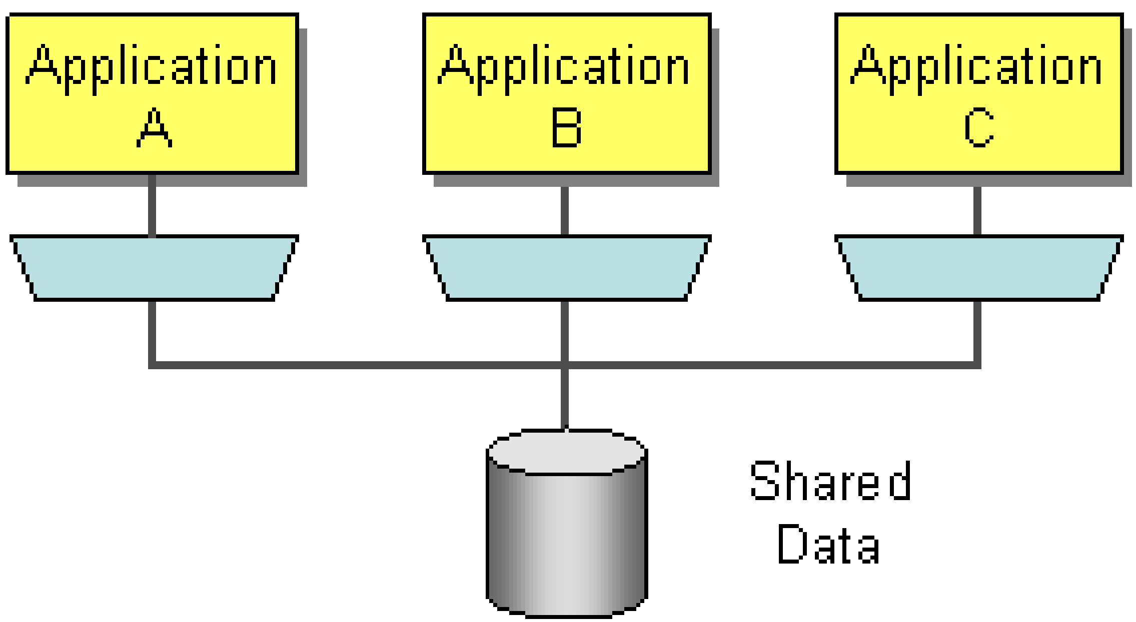
- filters must be independent entities
  - in particular, they should not share state with other filters.
- filters do not know the identity of their upstream and downstream filters.
- Their specifications might restrict what appears on the input pipes or make guarantees about what appears on the output pipes,
  - but they may not identify the components at the ends of those pipes.
- The correctness of the output of a pipe and filter network should not depend on the order in which the filters perform their incremental processing

Pipes and Filters

- Allow the designer to understand the overall behavior of a system
  - simple composition of the behaviors of the individual filters.
- Support reuse:
  - any filters can be connected together,
    - if they agree on the data that is being transmitted between them.
- Systems can be easily maintained and enhanced:
  - new filters can be added to existing systems
  - old filters can be replaced by improved ones.
- Allows specialized analysis
  - such as throughput and deadlock analysis.
- Naturally support concurrent execution.
  - Each filter can be implemented as a separate task and potentially executed in parallel with other filters.

Repositories

- Include two kinds of components:
  - a central data structure represents the current state
  - a collection of independent components that operate on the central data store



Repositories

- Repositories are bridges between data and operations that are in different domains.
- Provide consistency
  - If several application accesses the same data
- Provides transaction management
- Allows client interoperability
  - clients can be writen in multiple technologies
  - they only needs to use the catalog repository interface.

Repositories

- Can be implemented as a database
- can be implemented as a file system
- can be implemented as a cache

Event-based, Implicit Invocation

- Traditionally, systems provide a collection of procedures and functions
  - components interact with each other by explicitly invoking those routines
- In real-time/asynchronous/parallel systems
  - procedures are hard to handle
- Implici invocation based on events eases the development of such systems

Event-based, Implicit Invocation

- instead of invoking a procedure directly,
  - a component can announce (or broadcast) one or more events
  - expect for the event to be processed
- Other components in the system can register an interest in an event
  - by associating a procedure with the event
- event announcement
  - asynchronously
  - implicitly causes the invocation of procedures in other modules

Event-based, Implicit Invocation

- provides strong support for reuse.
  - Any component can be introduced into a system simply by registering it for the events of that system.
- Implicit invocation eases system evolution
  - Components may be replaced by other components without affecting the interfaces of other components in the system.
- Good for parallel systems

Event-based, Implicit Invocation

- Components relinquish control over the computation performed by the system.
  - When a component announces an event, it has no idea what other components will respond to it.
  - It know when they are finished.
- Data exchange
  - Data can be passed with the event.
  - But in other situations event systems must rely on a shared repository for interaction.
- Reasoning about correctness can be problematic
  - The meaning of a procedure that announces events will depend on the context of bindings in which it is invoked.
  - The order of the processing of events is not kown