



SS ZG653 (RL1.2): Software Architecture

A Brief History of Software Architecture

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Instructor: Prof. Santonu Sarkar

Informally what is meant by (Software) Architecture

- Essentially a blueprint of a software system that helps **stakeholders** to understand how the system would be once it is implemented
- What's should be there in this blueprint?
 - A description at a higher level of abstraction than objects and lines of codes

So that

- Stakeholders understand and reason about without getting lost into a sea of details

Who are Stakeholders?

A complex software has multiple stakeholders who expect certain features of the software

Stakeholder	Area of Concern
Chief Technologist	<ul style="list-style-type: none"> • Does it adhere to organization standards ?
Database Designer	<ul style="list-style-type: none"> • What information to be stored, where, how, access mechanism??? • Information security issues?
Application Development team	<ul style="list-style-type: none"> • How do I implement a complex scenario? • How should I organize my code? • How do I plan for division of work?
Users/Customers	<ul style="list-style-type: none"> • Does it perform as per my requirement? • What about the cost/budget? • Scalability, performance and reliability of the system? • How easy it is to use? • Is it always available?
Infrastructure Manager	<ul style="list-style-type: none"> • Performance and scalability • Idea of system & network usage • Indication of hardware and software cost, scalability, deployment location • Safety and security consideration • Is it fault tolerant-crash recovery & backup
Release & Configuration Manager	<ul style="list-style-type: none"> • Build strategy • Code management, version control, code organization
System Maintainer	<ul style="list-style-type: none"> • How do I replace of a subsystem with minimal impact ? • How fast can I diagnosis of faults and failures and how quickly I can recover?

Why Architecture needs to be described?

Any Large Software Corporation

- ❑ Hundreds of concurrent projects being executed
 - 10-100 team size
- ❑ Projects capture requirements, there are architects, and large Development teams
- ❑ Architect start with requirements team & handover to Development teams

- Each stakeholder has his own interpretation of the systems
 - Sometimes no understanding at all
 - Architect is the middleman who coordinates with these stakeholders
- How will everyone be convinced that his expectations from the system will be satisfied?
- Even when the architect has created the solution blueprint, how does she handover the solution to the developers?
- How do the developers build and ensure critical aspects of the system?
- Misunderstanding leads to incorrect implementation
 - Leads to 10 times more effort to fix at a later stage

Software Architecture Definition

- No unique definition though similar...
 - (look at <http://www.sei.cmu.edu/architecture/start/glossary/classicdefs.cfm>)
- .. “**structure** or structures of the system, which comprise **software elements**, the **externally visible properties** of those elements, and the **relationships** among them”

(Bass, Clements and Kazman, Software Architecture in Practice, 2nd edition)

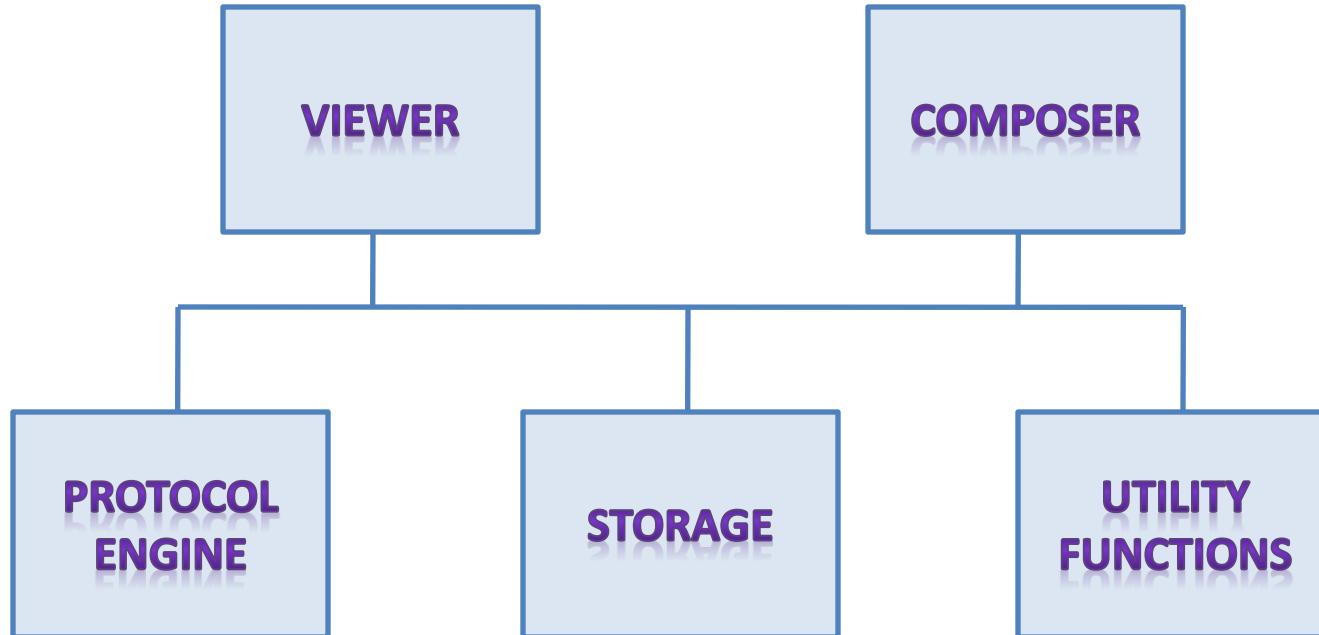
- “description of elements from which systems are built, **interactions** among those elements, **patterns** that guide their **composition**, and **constraints** on these patterns. In general, a particular system is defined in terms of a collection of **components** and interactions among these components”

Shaw and Garlan “Software Architecture: Perspectives on an Emerging Disciplines”

- “description of the **subsystems** and **components** of a software system and the **relationship** between them. Subsystems and components are typically specified in different **views** to show the relevant **functional** and **nonfunctional** properties of a software system”

F. Buschmann et al, Pattern Oriented Software Architecture

Is this Architecture



What we understand

- The system has 5 elements
- They are interconnected
- One is on the top of another

Typically we describe architecture as a collection of diagrams like this

What's Ambiguous?

- Visible responsibilities
 - What do they do?
 - How does their function relate to the system
 - How have these elements been derived, is there any overlap?
 - Are these processes, or programs
 - How do they interact when the software executes
 - Are they distributed?
 - How are they deployed on a hardware
 - What information does the system process?
-

What's Ambiguous?

- Significance of connections
 - Signify control or data, invoke each other, synchronization
 - Mechanism of communications
- Significance of layout
 - Does level shown signify anything
 - Was the type of drawing due to space constraint

What should Architecture description have?

- A structure describing
 - Modules
 - Services offered by each module
 - and their interactions- to achieve the functionality
 - Information/data modeling
 - Achieving quality attributes
 - Processes and tasks that execute the software
 - Deployment onto hardware
 - Development plan

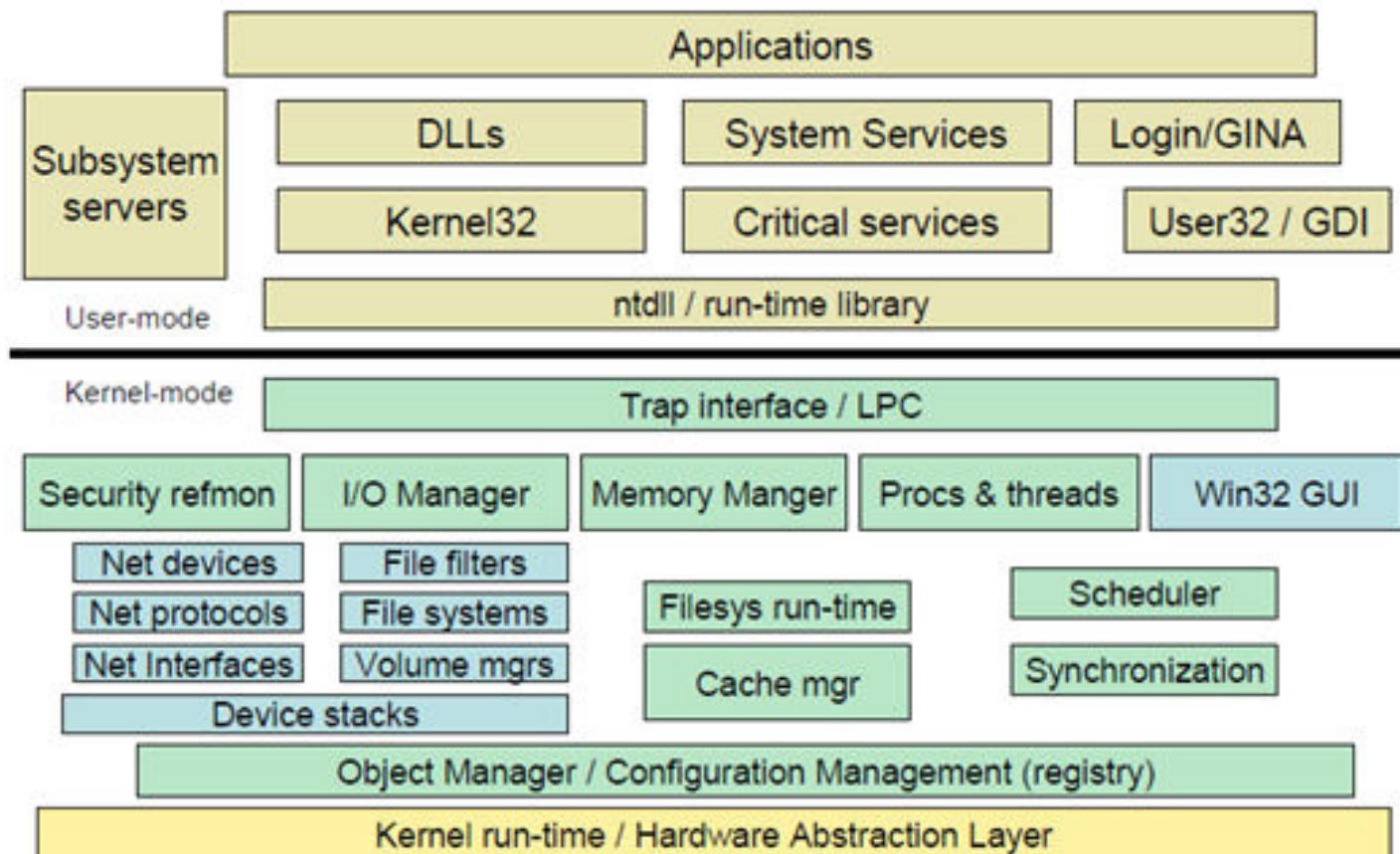
What should Architecture description have?.....

- A behavioral description
 - describing how the structural elements execute “important” and “critical” scenarios
 - E.g. how does the system authenticates a mobile user
 - How does the system processes 1 TB of data in a day
 - How does it stream video uninterrupted during peak load
 - These scenarios are mainly to implement various quality attributes

Architecture of Windows

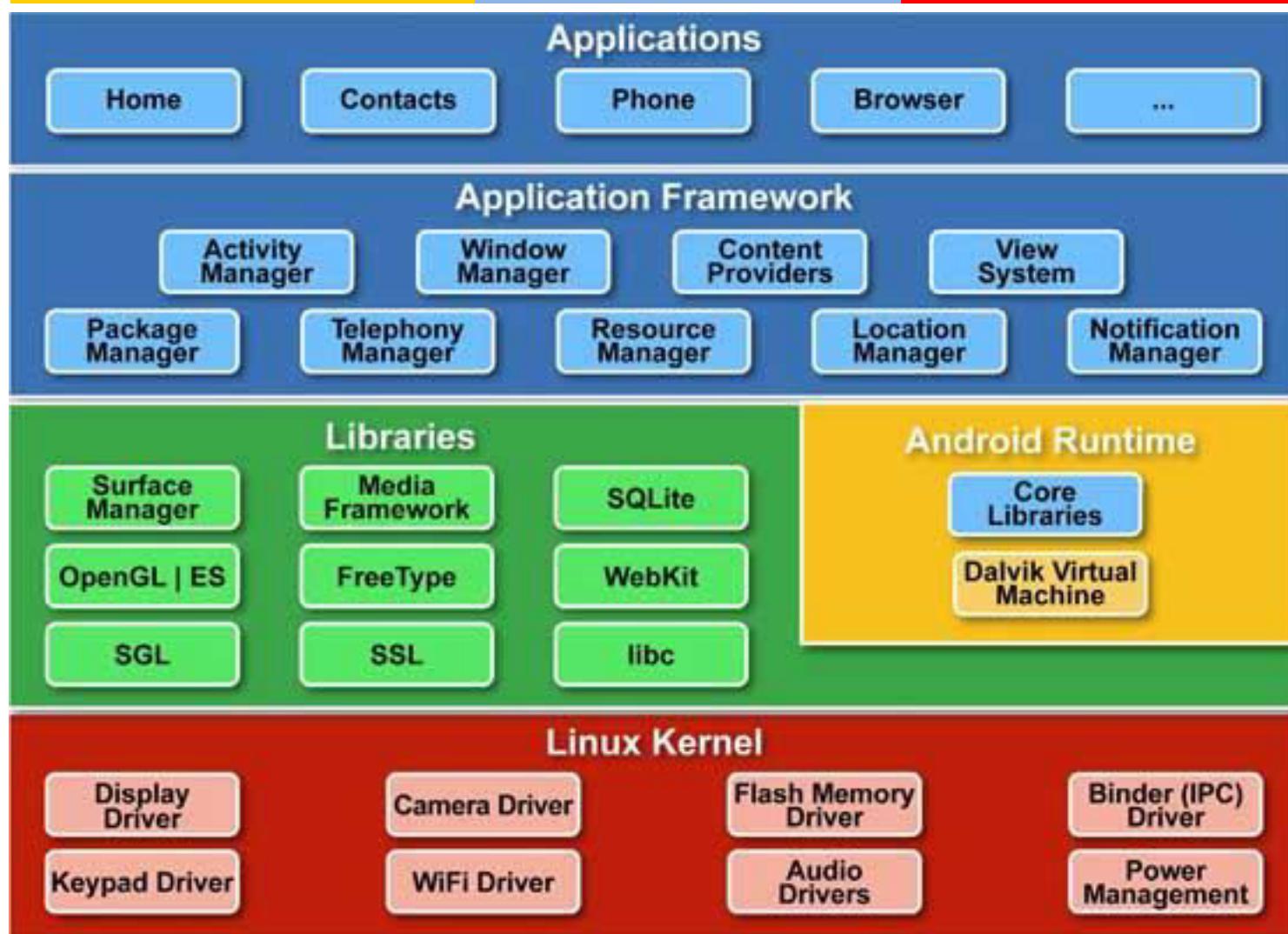
<https://blogs.msdn.com/b/hanybarakat/archive/2007/02/25/deeper-into-windows-architecture.aspx>

Windows Architecture



Architecture of Android

http://www.techotopia.com/index.php/An_Overview_of_the_Android_Architecture





SS ZG653 (RL 1.3): Software Architecture

Architecture Styles and Views

Instructor: Prof. Santonu Sarkar



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Architecture Styles

- Architecture style first proposed by Shaw and Garlan—synonymous to “architecture pattern”
 - A set of element types (what the element does- data store, compute linear regression function)
 - A set of interaction types (function call, publish-subscribe)
 - Topology indicating interactions and interaction types
 - Constraints
 - Also known as architectural pattern
- We shall cover some of these patterns in details

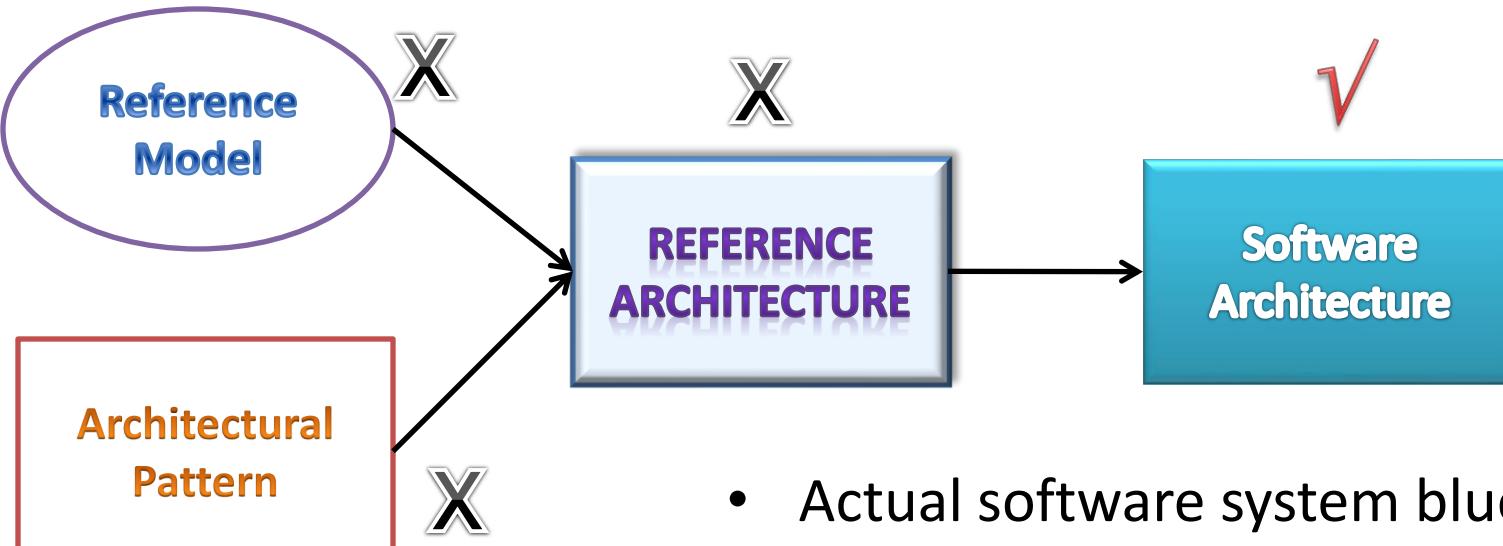
Views and Architectural Structure

- Since architecture serves as a vehicle for communication among stakeholders
 - And each stakeholder is interested about different aspects of the system
 - It is too complex to describe, understand and analyze the architecture using one common vocabulary for all stakeholders
 - Essentially it needs to be described in a multi-dimensional manner
- View based approach
 - Each view represents certain architectural aspects of the system, created for a stakeholder
 - All the views combined together form the consistent whole
- A Structure is the underlying part of a view- essentially the set of elements, and their properties
 - A view corresponding to a structure is created by using these elements and their inter-relationships

Reference Model and Reference Architecture

- A reference model
 - Decomposes the functionality into a set of smaller units
 - How they interact and share data
 - These units co-operatively implement the total functionality
- A reference architecture
 - Derived from the reference model
 - Concrete software elements, mapped to the units of the reference model, that implement the functionality

Inter-relationships



- Not architecture by itself!!
- Actual software system blueprint derived from requirement
- Contains design decisions
- Describes how it is deployed
- Addresses Quality of Service concerns

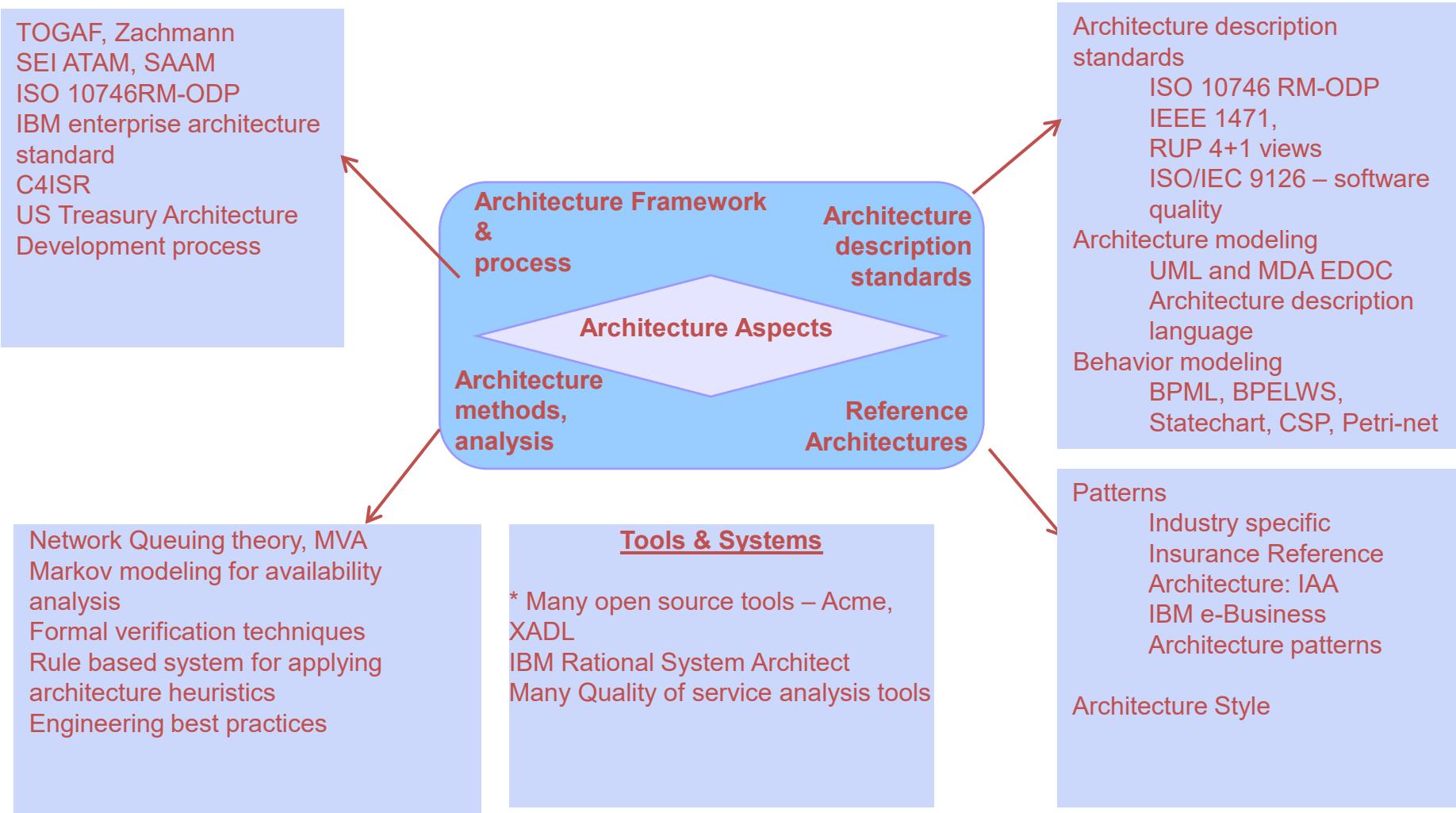
Benefits of Software Architecture

1. Every stakeholder should understand “unambiguously” what the blueprint is
 - Standard approach, vocabulary, output
 - Common language for communication
 2. Streamlining work assignments for multiple teams
 - Avoiding information loss, enforcing traceability
 3. Design decisions are made early
 - Quicker to evaluate these decisions and correct it rather than discovering it later (10 – 100 times more costly)
 - Early analysis of QoS and evaluation of architecture
 - Early analysis of meeting quality requirements and compromise between different QoS requirements
 - Early prototyping of important aspects quickly
 - More accurate cost and schedule estimation
 4. Improve speed of development
 - Reuse
 - Helps in building a large product line faster by sharing common architecture
 - From one implementation to another similar implementation
 - Based on the architecture, one can quickly decide build-vs –use external components
 - Tool that can automate part of development, testing
-

Three Structures will be covered

- Module Structure
 - How is the system to be structured as a set of code units (modules)?
- Component-and-connector structures
 - How is the system to be structured as a set of elements that have runtime behavior (components) and interactions (connectors)
 - What are major executing components and how do they interact
- Allocation structures
 - How is the system to relate to non-software structures in its environment (CPU or cluster of CPUs, File Systems, Networks, Development Teams ...)

In Bits and Pieces (Unfortunately)



Thank You



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 3.1): Software Architecture

Quality classes and attribute, quality attribute scenario and architectural tactics

Instructor: Prof. Santonu Sarkar

A step back

- What is functionality?
 - Ability of the system to fulfill its responsibilities
- Software Quality Attributes- also called non-functional properties
 - Orthogonal to functionality
 - is a constraint that the system must satisfy while delivering its functionality
- Design Decisions
 - A constraint driven by external factors (use of a programming language, making everything service oriented)

Consider the following requirements

- User interface should be easy to use
 - Radio button or check box? Clear text? Screen layout? --- NOT architectural decisions
 - User interface should allow redo/undo at any level of depth
 - Architectural decision
 - The system should be modifiable with least impact
 - Modular design is must – Architectural
 - Coding technique should be simple – not architectural
 - Need to process 300 requests/sec
 - Interaction among components, data sharing issues--architectural
 - Choice of algorithm to handle transactions -- non architectural
-

Quality Attributes and Functionality

- Any product (software products included) is sold based on its functionality – which are its features
 - Mobile phone, MS-Office software
 - Providing the desired functionality is often quite challenging
 - Time to market
 - Cost and budget
 - Rollout Schedule
- Functionality DOES NOT determine the architecture. If functionality is the only thing you need
 - It is perfectly fine to create a monolithic software blob!
 - You wouldn't require modules, threads, distributed systems, etc.

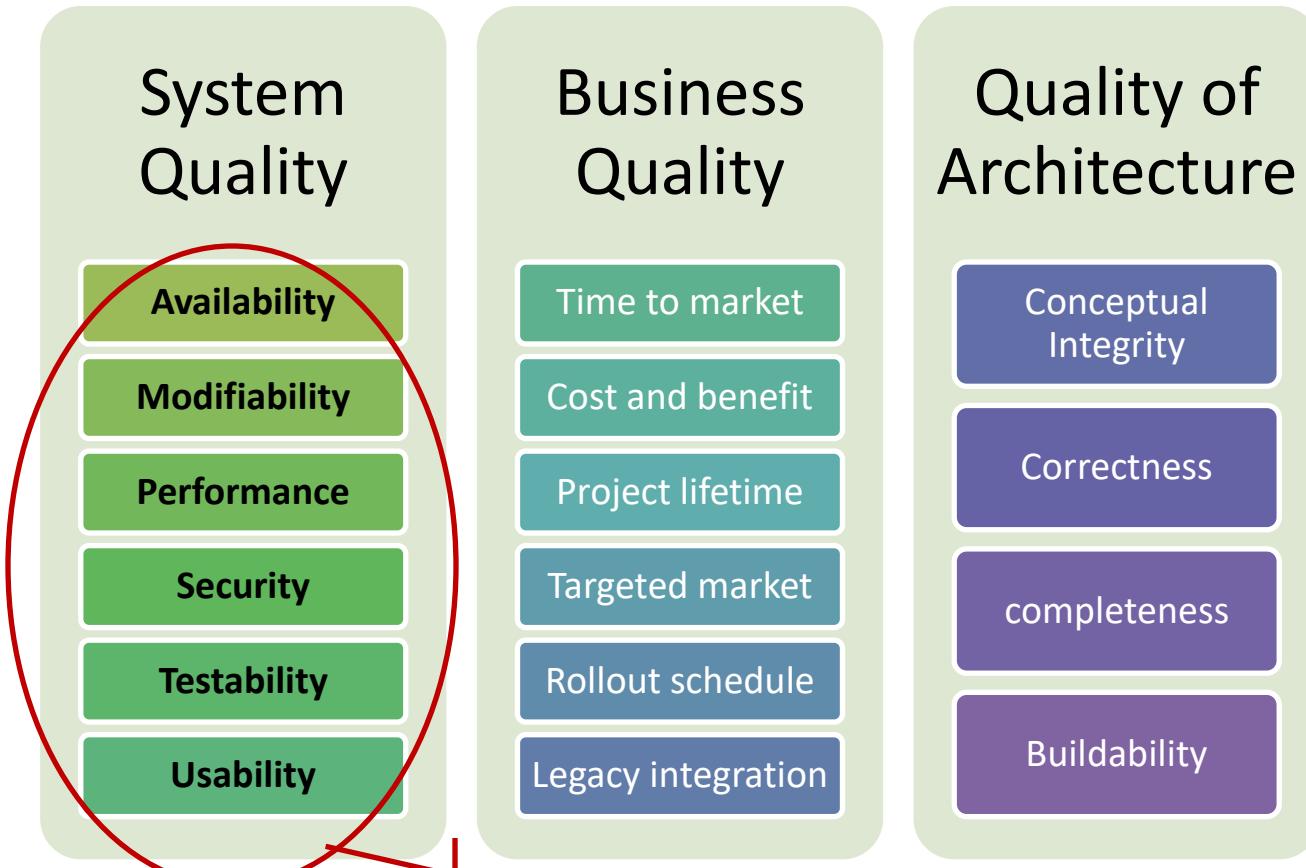
Examples of Quality Attributes

- Availability
 - Performance
 - Security
 - Usability
 - Functionality
 - Modifiability
 - Portability
 - Reusability
 - Integrability
 - Testability
- The success of a product will ultimately rest on its Quality attributes
 - “Too slow!” -- performance
 - “Keeps crashing!” --- availability
 - “So many security holes!” --- security
 - “Reboot every time a feature is changed!” --- modifiability
 - “Does not work with my home theater!” --- integrability
 - Needs to be achieved throughout the design, implementation and deployment
 - Should be designed in and also evaluated at the architectural level
 - Quality attributes are NON-orthogonal
 - One can have an effect (positive or negative) on another
 - Performance is troubled by nearly all other. All other demand more code whereas performance demands the least

Defining and understanding system quality attributes

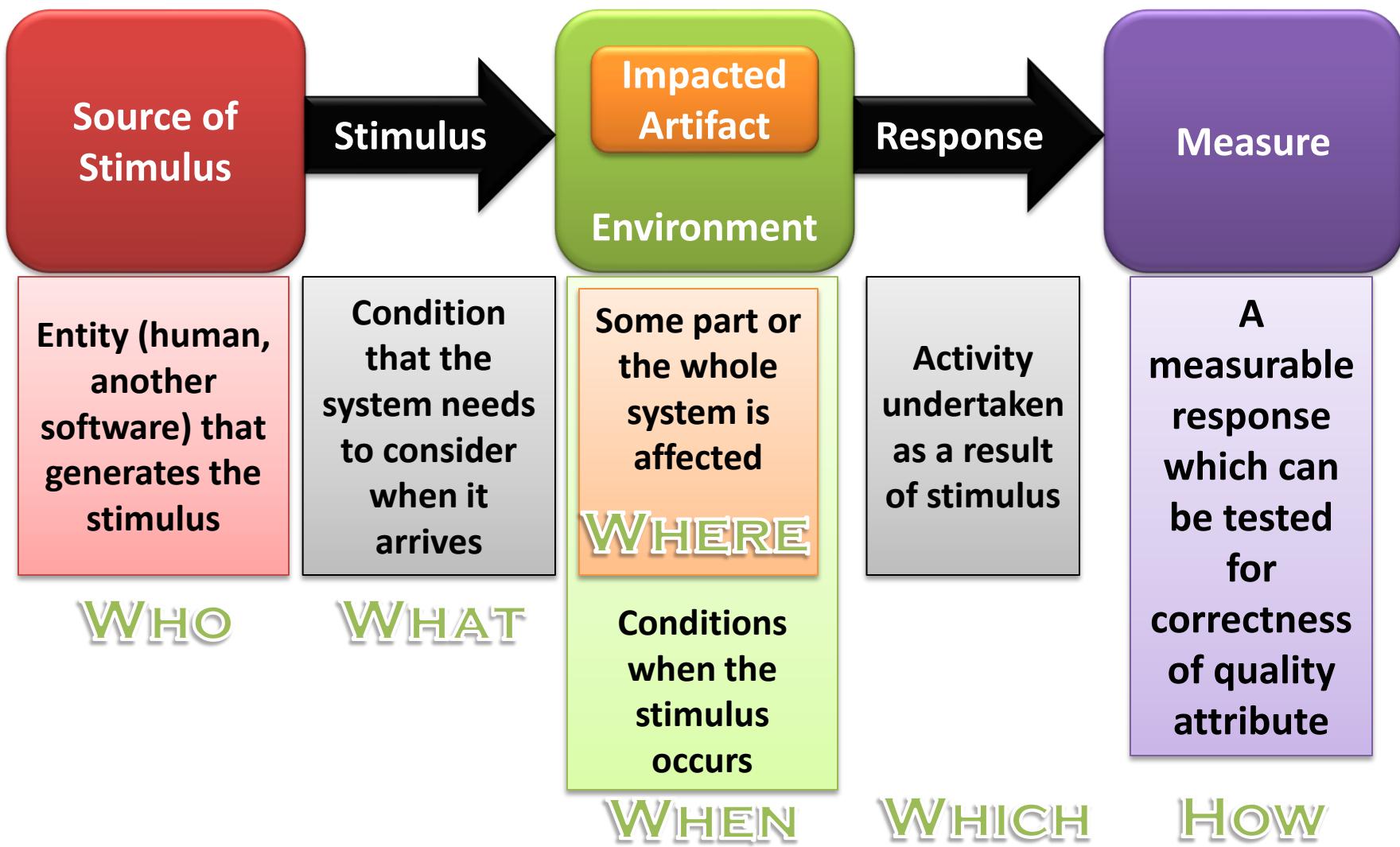
- Defining a quality attribute for a system
 - System should be modifiable --- vague, ambiguous
 - How to associate a failure to a quality attribute
 - Is it an availability problem, performance problem or security or all of them?
 - Everyone has his own vocabulary of quality
 - ISO 9126 and ISO 25000 attempts to create a framework to define quality attributes
-

Three Quality Classes



- We will consider these attributes
- We will use “**Quality Attribute Scenarios**” to characterize them
 - which is a quality attribute specific requirement

Quality Attribute Scenario



Architectural Tactics

- To achieve a quality one needs to take a design decision- called Tactic
 - Collection of such tactics is **architectural strategy**
 - A pattern can be a collection of tactics



Quality Design Decisions

- To address a quality following 7 design decisions need to be taken
 - Allocation of responsibilities
 - Coordination
 - Data model
 - Resource Management
 - Resource Binding
 - Technology choice

Quality Design Decisions

- **Responsibility Allocation**
 - Identify responsibilities (features) that are necessary for this quality requirement
 - Which non-runtime (module) and runtime (components and connectors) should address the quality requirement
- **Coordination**
 - Mechanism (stateless, stateful...)
 - Properties of coordination (lossless, concurrent etc.)
 - Which element should and shouldn't communicate
- **Data Model**
 - What's the data structure, its creation, use, persistence, destruction mechanism
 - Metadata
 - Data organization
- **Resource management**
 - Identifying resources (CPU, I/O, memory, battery, system lock, thread pool..) and who should manage
 - Arbitration policy
 - Find impact of what happens when the threshold is exceeded
- **Binding time decision**
 - Use parameterized makefiles
 - Design runtime protocol negotiation during coordination
 - Runtime binding of new devices
 - Runtime download of plugins/apps
- **Technology choice**

Business Qualities

Business Quality	Details
Time to Market	<ul style="list-style-type: none"> • Competitive Pressure – short window of opportunity for the product/system • Build vs. Buy decisions • Decomposition of system – insert a subset OR deploy a subset
Cost and benefit	<ul style="list-style-type: none"> • Development effort is budgeted • Architecture choices lead to development effort • Use of available expertise, technology • Highly flexible architecture costs higher
Projected lifetime of the system	<ul style="list-style-type: none"> • The product that needs to survive for longer time needs to be modifiable, scalable, portable • Such systems live longer; however may not meet the time-to-market requirement
Targeted Market	<ul style="list-style-type: none"> • Size of potential market depends on feature set and the platform • Portability and functionality key to market share • Establish a large market; a product line approach is well suited
Rollout Schedule	<ul style="list-style-type: none"> • Phased rollouts; base + additional features spaced in time • Flexibility and customizability become the key
Integration with Legacy System	<ul style="list-style-type: none"> • Appropriate integration mechanisms • Much implications on architecture

Architectural Qualities

Architectural Quality	Details
Conceptual Integrity	<ul style="list-style-type: none">•Architecture should do similar things in similar ways•Unify the design at all levels
Correctness and Completeness	<ul style="list-style-type: none">•Essential to ensure system's requirements and run time constraints are met
Build ability	<ul style="list-style-type: none">•Implemented by the available team in a timely manner with high quality•Open to changes or modifications as time progresses•Usually measured in cost and time•Knowledge about the problem to be solved



SS ZG653 (RL 4.1): Software Architecture

Usability and Its Tactics

Instructor: Prof. Santonu Sarkar



BITS Pilani

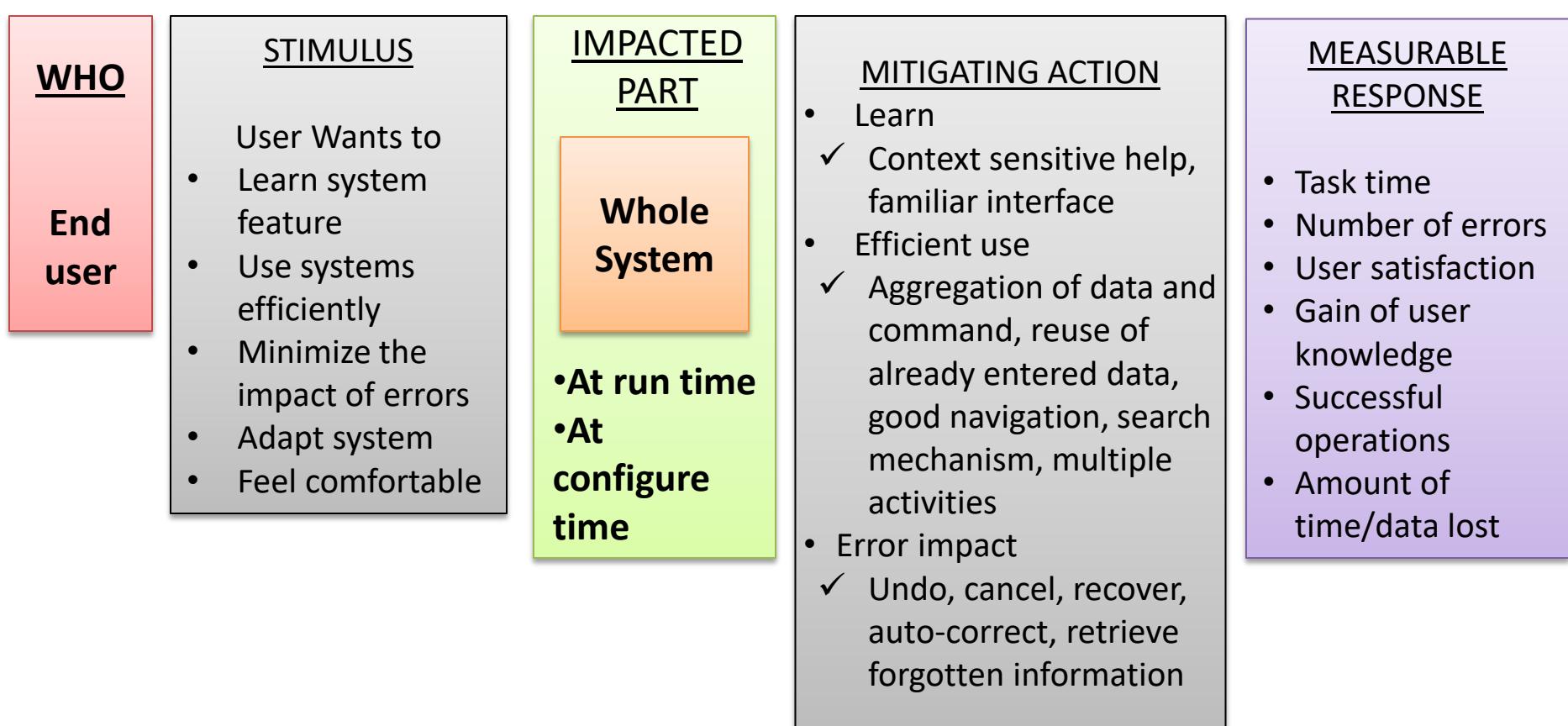
Pilani|Dubai|Goa|Hyderabad

Usability

- How easy it is for the user to accomplish a desired task and user support the system provides
 - Learnability: what does the system do to make a user familiar
 - Operability:
 - Minimizing the impact of user errors
 - Adopting to user needs
 - Giving confidence to the user that the correct action is being taken?



Usability Scenario Example



End User

Downloads application

Runtime

Uses application productively

Takes 4 mins to be productive

Usability Tactics

Usability is essentially Human Computer Interaction. Runtime Tactics are

User initiative
(and system responds)

System initiative

Cancel, undo,
aggregation, store
partial result

Task model:
understands the
context of the task
user is trying and
provide assistance

User model:
understands who
the user is and
takes action

System model:
gets the current
state of the system
and responds

User Initiative and System



Response

- Cancel
 - When the user issues cancel, the system must listen to it (in a separate thread)
 - Cancel action must clean the memory, release other resources and send cancel command to the collaborating components
- Undo
 - System needs to maintain a history of earlier states which can be restored
 - This information can be stored as snapshots
- Pause/resume
 - Should implement the mechanism to temporarily stop a running activity, take its snapshot and then release the resource for other's use
- Aggregate (change font of the entire paragraph)
 - For an operation to be applied to a large number of objects
 - Provide facility to group these objects and apply the operation to the group

System Initiated

- Task model
 - Determine the current runtime context, guess what user is attempting, and then help
 - Correct spelling during typing but not during password entry
- System model
 - Maintains its own model and provide feedback of some internal activities
 - Time needed to complete the current activity
- User model
 - Captures user's knowledge of the system, behavioral pattern and provide help
 - Adjust scrolling speed, user specific customization, locale specific adjustment

Usability Tactics and Patterns....

- Design time tactics- UI is often revised during testing. It is best to separate UI from the rest of the application
 - Model view controller architecture pattern
 - Presentation abstraction control
 - Command Pattern
 - Arch/Slinky
 - Similar to Model view controller

Design Checklist

- Allocation of Responsibilities
 - Identify the modules/components responsible for
 - Providing assistance, on-line help
 - Adapt and configure based on user choice
 - Recover from user error
- Coordination Model
 - Check if the system needs to respond to
 - User actions (mouse movement) and give feedback
 - Can long running events be canceled?
- Data model
 - data structures needed for undo, cancel
 - Design of transaction granularity to support undo and cancel
- Resource mgmt
 - Design how user can configure system's use of resource
- Technology selection
 - To achieve usability

Thank You



BITS Pilani

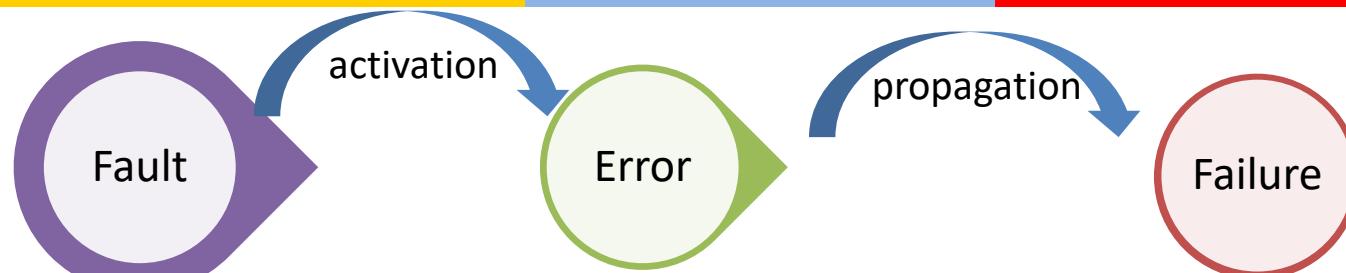
Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 4.2): Software Architecture

Availability and Its Tactics

Instructor: Prof. Santonu Sarkar

Faults and Failure



- Hypothesized cause of error in the software
- Part of the system's total state that can leads to failure
- event that occurs when the delivered service deviates from correct service
- Not every fault causes a failure:
 - Code that is “mostly” correct.
 - Dead or infrequently-used code.
 - Faults that depend on a set of circumstances to occur
- Cost of software failure often far outstrips the cost of the original system
 - data loss
 - down-time
 - cost to fix
- **Primary objective:** Remove faults with the most serious consequences.
- **Secondary objective:** Remove faults that are encountered most often by users.
 - One study showed that removing 60% of software “defects” led to a 3% reliability improvement

Failure Classification

- Transient - only occurs with certain inputs
 - Permanent - occurs on all inputs
 - Recoverable - system can recover without operator help
 - Unrecoverable - operator has to help
 - Non-corrupting - failure does not corrupt system state or data
 - Corrupting - system state or data are altered
-

Availability

- Readiness of the software to carry out its task
 - 100% available (which is actually impossible) means it is always ready to perform the intended task
 - A related concept is Reliability
 - Ability to “continuously provide” correct service without failure
 - Availability vs Reliability
 - A software is said to be available even when it fails but recovers immediately
 - Such a software will NOT be called Reliable
 - Thus, Availability measures the fraction of time system is really available for use
 - Takes repair and restart times into account
 - Relevant for non-stop continuously running systems (e.g. traffic signal)
-

What is Software Reliability

- Probability of failure-free operation of a system over a specified time within a specified **environment** for a specified **purpose**
 - Difficult to measure the **purpose**,
 - Difficult to measure **environmental factors**.
- It's not enough to consider simple failure rate:
 - Not all failures are created equal; some have much more serious consequences.
 - Might be able to recover from some failures reasonably.

Availability

- Once the system fails
 - It is not available
 - It needs to recover within a short time
- Availability 
- Scheduled downtime is typically not considered
 - Availability 100% means it recovers instantaneously
 - Availability 99.9% means there is 0.01% probability that it will not be operational when needed

System Type	Availability (%)	Downtime in a year
Normal workstation	99	3.6 days
HA system	99.9	8.5 hours
Fault-resilient system	99.99	1 hour
Fault-tolerant system	99.999	5 min

Availability Scenarios

<u>WHO</u> Internal or External to System	<u>STIMULUS</u> Fault causing <ul style="list-style-type: none"> ➤ System does not respond ➤ Crash ➤ Delay in response ➤ Erroneous Response 	<u>IMPACTED PART</u> Infrastructure and/or application <ul style="list-style-type: none"> • During normal operation • During degraded mode of operation 	<u>MITIGATING ACTION</u> When fault occurs it should do one or more of <ul style="list-style-type: none"> ✓ detect and log ✓ Notify the relevant stakeholders ✓ Disable the source of failure ✓ Be unavailable for a predefined time interval ✓ Continue to operate in a degraded mode 	<u>MEASURABLE RESPONSE</u> <ul style="list-style-type: none"> • Specific time interval for availability • Availability number • Time interval when it runs in degraded mode • Time to repair
--	--	---	--	---

Two Broad Approaches

- Fault Tolerance
 - Allow the system to continue in presence of faults.
Methods are
 - Error Detection
 - Error Masking (through redundancy)
 - Recovery
 - Fault Prevention
 - Techniques to avoid the faults to occur
-

Availability Tactics

Fault detection	Error Masking	Recover From Fault	Fault prevention
<ul style="list-style-type: none"> • Ping/echo • Heartbeat • Timestamp • Data sanity check • Condition monitoring • Voting • Exception Detection • Self-test 	<ul style="list-style-type: none"> • Active redundancy (Hot) • Passive redundancy (Warm) • Spare (Cold) • Exception handling • Graceful degradation • Ignore faulty behavior 	<ul style="list-style-type: none"> • Rollback • Retry • Reconfiguration • Shadow operation • State resynchronization • Escalating restart • Nonstop forwarding 	<ul style="list-style-type: none"> • Removal of a component to prevent anticipated failure—auto/manual reboot • Create transaction • Software upgrade • Predictive model • Process monitor—that can detect, remove and restart faulty process • Exception prevention

Availability Tactics- Fault Detection

- Ping
 - Client (or fault-detector) pings the server and gets response back
 - To avoid less communication bandwidth- use hierarchy of fault-detectors, the lowest one shares the same h/w as the server
- Heartbeat
 - Server periodically sends a signal
 - Listeners listen for such heartbeat. Failure of heartbeat means that the server is dead
 - Signal can have data (ATM sending the last txn)
- Exception Detection
 - Adding an Exception handler means error masking

More details- Heartbeat

- Each node implements a lightweight process called heartbeat daemon that periodically (say 10 sec) sends heartbeat message to the master node.
 - If master receives heartbeat from a node from both connections (a node is connected redundantly for fault-tolerance), everything is ok
 - If it gets from one connections, it reports that one of the network connection is faulty
 - If it does not get any heartbeat, it reports that the node is dead (assuming that the master gets heartbeat from other nodes)
 - Trick: Often heartbeat signal has a payload (say resource utilization info of that node)
 - Hadoop NameNode uses this trick to understand the progress of the job
-

Detect Fault

- Timer and Timestamping
 - If the running process does not reset the timer periodically, the timer triggers off and announces failure
 - Timestamping: assigns a timestamp (can be a count, based on the local clock) with a message in a decentralized message passing system. Used to detect inconsistency
 - Voting (TMR)
 - Three identical copies of a module are connected to a voting system which compares outputs from all the three components. If there is an inconsistency in their outputs when subjected to the same input, the voting system reports error/inconsistency
 - Majority voting, or preferred component wins
-

Availability Tactics- Error Masking

- Hot spare (Active redundancy)
 - Every redundant process is active
 - When one fails, another one is taken up
 - Downtime is millisec
- Warm restart (Passive redundancy)
 - Standbys keep syncing their states with the primary one
 - When primary fails, backup starts
- Spare copy (Cold)
 - Spares are offline till the primary fails, then it is restarted
 - Typically restarts to the checkpointed position
 - Downtime in minute
 - Used when the MTTF is high and HA is not that critical

Error Masking

- Service Degradation
 - Most critical components are kept live and less critical component functionality is dropped
- Ignore faulty behavior
 - E.g. If the component send spurious messages or is under DOS attack, ignore output from this component
- Exception Handling – this masks or even can correct the error

Availability Tactics- Fault Recovery

- Shadow
 - Repair the component
 - Run in shadow mode to observe the behavior
 - Once it performs correctly, reintroduce it
 - State resynch
 - Related to the hot and warm restart
 - When the faulty component is started, its state must be upgraded to the latest state.
 - Update depends on downtime allowed, size of the state, number of messages required for the update..
 - Checkpointing and recovery
 - Application periodically “commits” its state and puts a checkpoint
 - Recovery routines can either roll-forward or roll-back the failed component to a checkpoint when it recovers
-

Availability Tactics- Recovery

- Escalating Restart
 - Allows system to restart at various levels of granularity
 - Kill threads and recreate child processes
 - Frees and reinitialize memory locations
 - Hard restart of the software
 - Nonstop forwarding (used in router design)
 - If the main recipient fails, the alternate routers keep receiving the packets
 - When the main recipient comes up, it rebuilds its own state
-

Availability Tactics- Fault Prevention

- Faulty component removal
 - Fault detector predicts the imminent failure based on process's observable parameters (memory leak)
 - The process can be removed (rebooted) and can be auto-restart
- Transaction
 - Group relevant set of instructions to a transaction
 - Execute a transaction so that either everyone passes or all fails
- Predictive Modeling
 - Analyzes past failure history to build an empirical failure model
 - The model is used to predict upcoming failure
- Software upgrade (preventive maintenance)
 - Periodic upgrade of the software through patching prevents known vulnerabilities

Design Decisions

Responsibility Allocation

- For each service that need to be highly available
 - Assign additional responsibility for fault detection (e.g. crash, data corruption, timing mismatch)
 - Assign responsibilities to perform one or more of:
 - Logging failure, and notification
 - Disable source event when fault occur
 - Implement fault-masking capability
 - Have mechanism to operate on degraded mode

Coordination

- For each service that need to be highly available
 - Ensure that the coordination mechanism can sense the crash, incorrect time
 - Ensure that the coordination mechanism will
 - Log the failure
 - Work in degraded mode

Design Decisions

Data Model

- Identify which data + operations are impacted by a crash, incorrect timing etc.
 - Ensure that these data elements can be isolated when fault occurs
 - E.g. ensure that “write” req. is cached during crash so that during recovery these writes are applied to the system

Resource Management

- Identify which resources should be available to continue operations during fault
- E.g. make the input Q large enough so that can accommodate requests when the server is being recovered from a failure

Design Decisions

- Binding Time
 - Check if late binding can be a source of failure
 - Suppose that a late bound component report its failure in 0.1ms after the failure and the recovery takes 1.5sec. This may not be acceptable
- Technology Choice
 - Determine the technology and tools that can help in fault detection, recovery and then reintroduction
 - Determine the technology that can handle a fault
 - Determine whether these tools have high availability!!

Hardware vs Software Reliability



Metrics

- Hardware metrics are not suitable for software since its metrics are based on notion of component failure
 - Software failures are often design failures
 - Often the system is available after the failure has occurred
 - Hardware components can wear out
-

Software Reliability Metrics

- Reliability metrics are units of measure for system reliability
 - System reliability is measured by counting the number of operational failures and relating these to demands made on the system at the time of failure
 - A long-term measurement program is required to assess the reliability of critical systems
-

Time Units

- Raw Execution Time
 - non-stop system
 - Calendar Time
 - If the system has regular usage patterns
 - Number of Transactions
 - demand type transaction systems
-

Reliability Metric POFOD

- Probability Of Failure On Demand (POFOD):
 - Likelihood that system will fail when a request is made.
 - E.g., POFOD of 0.001 means that 1 in 1000 requests may result in failure.
- Any failure is important; doesn't matter how many if the failure > 0
- Relevant for safety-critical systems

Reliability Metric ROCOF & MTTF

- Rate Of Occurrence Of Failure (ROCOF):
 - Frequency of occurrence of failures.
 - E.g., ROCOF of 0.02 means 2 failures are likely in each 100 time units.
 - Relevant for transaction processing systems
 - Mean Time To Failure (MTTF):
 - Measure of time between failures.
 - E.g., MTTF of 500 means an average of 500 time units passes between failures.
 - Relevant for systems with long transactions
-

Rate of Fault Occurrence

- Reflects rate of failure in the system
 - Useful when system has to process a large number of similar requests that are relatively frequent
 - Relevant for operating systems and transaction processing systems
-

Mean Time to Failure

- Measures time between observable system failures
 - For stable systems $MTTF = 1/ROCOF$
 - Relevant for systems when individual transactions take lots of processing time (e.g. CAD or WP systems)
-

Failure Consequences

- When specifying reliability both the number of failures and the consequences of each matter
 - Failures with serious consequences are more damaging than those where repair and recovery is straightforward
 - In some cases, different reliability specifications may be defined for different failure types
-

Building Reliability Specification

- For each sub-system analyze consequences of possible system failures
 - From system failure analysis partition failure into appropriate classes
 - For each class send out the appropriate reliability metric
-

Examples

Failure Class	Example	Metric
Permanent Non-corrupting	ATM fails to operate with any card, must restart to correct	ROCOF = .0001 Time unit = days
Transient Non-corrupting	Magnetic stripe can't be read on undamaged card	POFOD = .0001 Time unit = transactions

THANK YOU

Reliability Metrics - part 1

- Probability of Failure on Demand (POFOD)
 - POFOD = 0.001
 - For one in every 1000 requests the service fails per time unit
 - Rate of Fault Occurrence (ROCOF)
 - ROCOF = 0.02
 - Two failures for each 100 operational time units of operation
-

Reliability Metrics - part 2

- Mean Time to Failure (MTTF)
 - average time between observed failures (aka MTBF)
 - Availability = MTTF / (MTTF+MTTR)
 - MTTF = Mean Time To Failure
 - MTTR = Mean Time to Repair
 - Reliability = MTBF / (1+MTBF)
-

Probability of Failure on Demand

- Probability that the system will fail when a service request is made
 - Useful when requests are made on an intermittent or infrequent basis
 - Appropriate for protection systems service requests may be rare and consequences can be serious if service is not delivered
 - Relevant for many safety-critical systems with exception handlers
-

Specification Validation

- It is impossible to empirically validate high reliability specifications
 - No database corruption really means POFOD class < 1 in 200 million
 - If each transaction takes 1 second to verify, simulation of one day's transactions takes 3.5 days
-

Statistical Reliability Testing

- Test data used, needs to follow typical software usage patterns
 - Measuring numbers of errors needs to be based on errors of omission (failing to do the right thing) and errors of commission (doing the wrong thing)
-

Difficulties with Statistical Reliability



Testing

- Uncertainty when creating the operational profile
 - High cost of generating the operational profile
 - Statistical uncertainty problems when high reliabilities are specified
-

Safety Specification

- Each safety specification should be specified separately
 - These requirements should be based on hazard and risk analysis
 - Safety requirements usually apply to the system as a whole rather than individual components
 - System safety is an emergent system property
-

THANK YOU



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 5.1): Software Architecture Modifiability and Its Tactics

Instructor: Prof. Santonu Sarkar

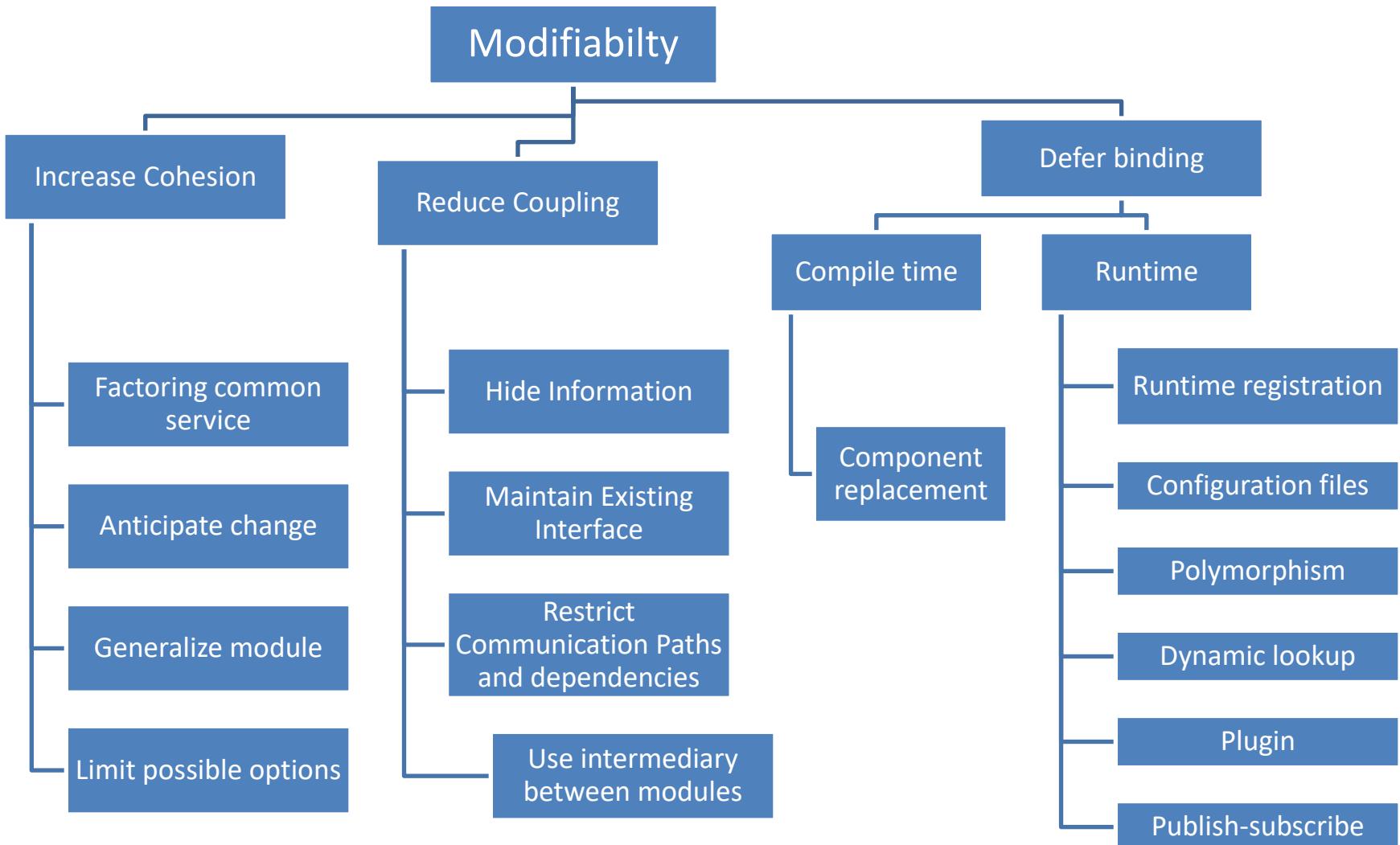
Modifiability

- Ability to Modify the system based on the change in requirement so that
 - the time and cost to implement is optimal
 - Impact of modification such as testing, deployment, and change management is minimal
- When do you want to introduce modifiability?
 - If $(\text{cost of modification w/o modifiability mechanism in place}) > (\text{cost of modification with modifiability in place}) + \text{Cost of installing the mechanism}$

Modifiability Scenarios

<u>WHO</u>	<u>STIMULUS</u>	<u>IMPACTED PART</u>	<u>MITIGATING ACTION</u>	<u>MEASURABLE RESPONSE</u>
<ul style="list-style-type: none"> • Enduser • Developer • SysAdm 	<p>They want to modify</p> <ul style="list-style-type: none"> ➢ Functionality <ul style="list-style-type: none"> ➢ Add, modify, delete ➢ Quality <ul style="list-style-type: none"> ➢ Capacity 	<p><u>UI, platform or System</u></p> <ul style="list-style-type: none"> • Runtime • Compile time • Design time • Build time 	<p>When fault occurs it should do one or more of</p> <ul style="list-style-type: none"> ✓ Locate (Impact analysis) ✓ Modify ✓ Test ✓ Deploy again 	<ul style="list-style-type: none"> • Volume of the impact of the primary system (number, size) • Cost of modification • Time and effort • Extent of impact to other systems • New defects introduced
Developer	Tries to change UI	Artifact – Code Environment: Design time	Changes made and unit test done	Completed in 4 hours

Modifiability Tactics



Dependency between two modules

(B → A)

Syntax (compile+runtime)

- Data : B uses the type/format of the data created by A
- Service B uses the API signature provided by A

Semantics of A

- Data: Semantics of data created by A should be consistent with the assumption made by B
- Service: Same

Sequence

- Data -- data packets created by A should maintain the order as understood by B
- Control– A must execute 5ms before B. Or an API of A can be called only after calling another API

Interface identity

- Handle of A must be consistent with B, if A maintains multiple interfaces

Location of A

- B may assume that A is in-process or in a different process, hardware..

Quality of service/data provided by A

- Data quality produced by A must be > some accuracy for B to work

Existence of A

- B may assume that A must exist when B is calling A

Resource behavior of A

- B may assume that both use same memory
- B needs to reserve a resource owned by A

Localize Modifications

1. Factoring common service

- Common services through a specialized module (only implementing module should be impacted)
 - Heavily used in application framework and middleware
- Reduce Coupling and increase cohesion

2. Anticipate Expected Changes

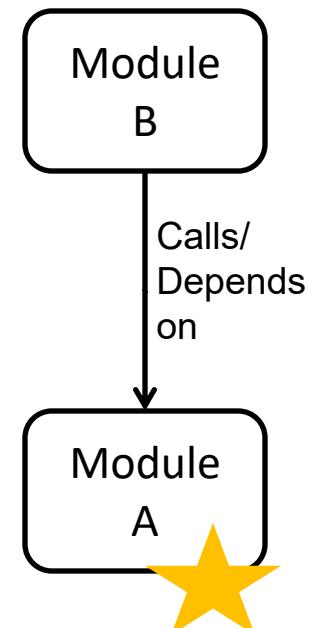
- Quite difficult to anticipate, hence should be coupled with previous one
- Allow extension points to accommodate changes

3. Generalize the Module

- Allowing it to perform broader range of functions
- Externalize configuration parameters (could be described in a language like XML)
 - The module reconfigure itself based on the configurable parameters
- Externalize business rules

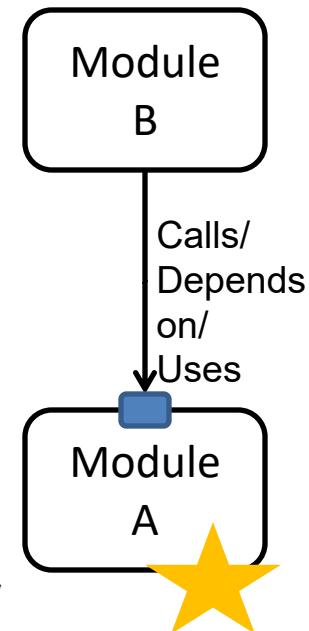
4. Limit Possible options

- Do not keep too many options for modules that are part of the framework



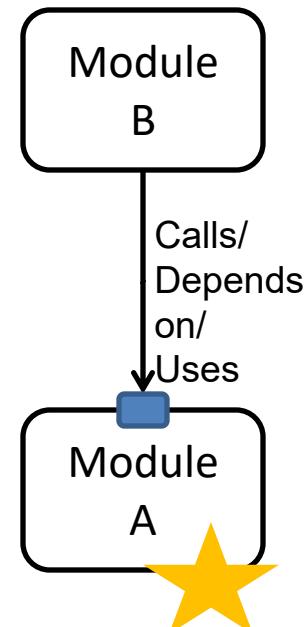
Prevent Ripple Effect Tactics

1. Hide Information (of A)
 - Use interfaces, allow published API based calls only
2. Maintain existing Interface (of A)
 - Add new interfaces if needed
 - Use Wrapper, adapter to maintain same interface
 - Use stub
3. Restrict Communication Paths
 - Too many modules should not depend on A
4. Use an intermediary between B and A
 - Data
 - Repository from which B can read data created by A (blackboard pattern)
 - Publish-subscribe (data flowing through a central hub)
 - MVC pattern
 - Service: Use of design patterns like bridge, mediator, strategy, proxy
 - Identity of A – Use broker pattern which deals with A's identity
 - Location of A – Use naming service to discover A
 - Existence of A- Use factory pattern



Defer Binding Time

1. Runtime registration of A (plug n play)
 - Use of pub-sub
2. Configuration Files
 - To take decisions during startup
3. Polymorphism
 - Late binding of method call
4. Component Replacement (for A)
 - during load time such as classloader
5. Adherence to a defined protocol- runtime binding of independent processes



Design Checklist- Modifiability

- Allocation of Responsibilities
 - Determine the types of changes that can come due to technical, customer or business
 - Determine what sort of additional features are required to handle the change
 - Determine which existing features are impacted by the change
- Coordination Model
 - For those where modifiability is a concern, use techniques to reduce coupling
 - Use publish-subscribe, use enterprise service bus
 - Identify
 - which features can change at runtime
 - which devices, communication paths or protocols can change at runtime
 - And make sure that such changes have limited impact on the system

Design checklist- Modifiability

- Data Model
 - For the anticipated changes, decide which data elements will be impacted, and the nature of impact (creation, modification, deletion, persistence, translation)
 - Group data elements that are likely to change together
 - Design to ensure that changes have minimal impact to the rest of the system
- Resource Management
 - Determine how addition, deletion or modification of a feature or a quality attribute cause
 - New resources to be used, or affect resource usage
 - Changing of resource usage limits
 - Ensure that the resources after modification are sufficient to meet the system requirement
 - Write Resource manager module that encapsulates resource usage policies

Design Checklist- Modifiability

- Binding
 - Determine the latest time at which the anticipated change is required
 - Choose a defer binding if possible
 - Try to avoid too many binding choices
 - Choice of Technology
 - Evaluate the technology that can handle modifications with least impact (e.g. enterprise service bus)
 - Watch for vendor lock-in
-



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 5.2): Software Architecture

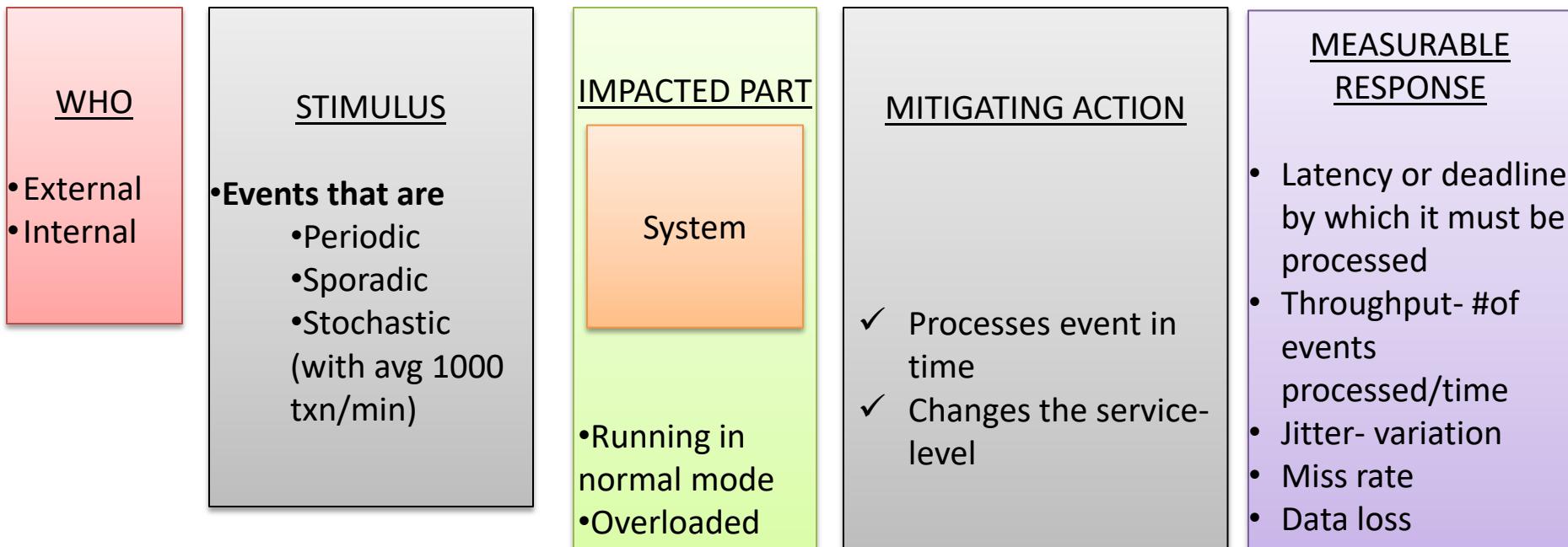
Performance and Its Tactics

Instructor: Prof. Santonu Sarkar

What is Performance?

- Software system's ability to meet timing requirements when it responds to an event
- Events are
 - interrupts, messages, requests from users or other systems
 - clock events marking the passage of time
- The system, or some element of the system, must **respond to them** in time

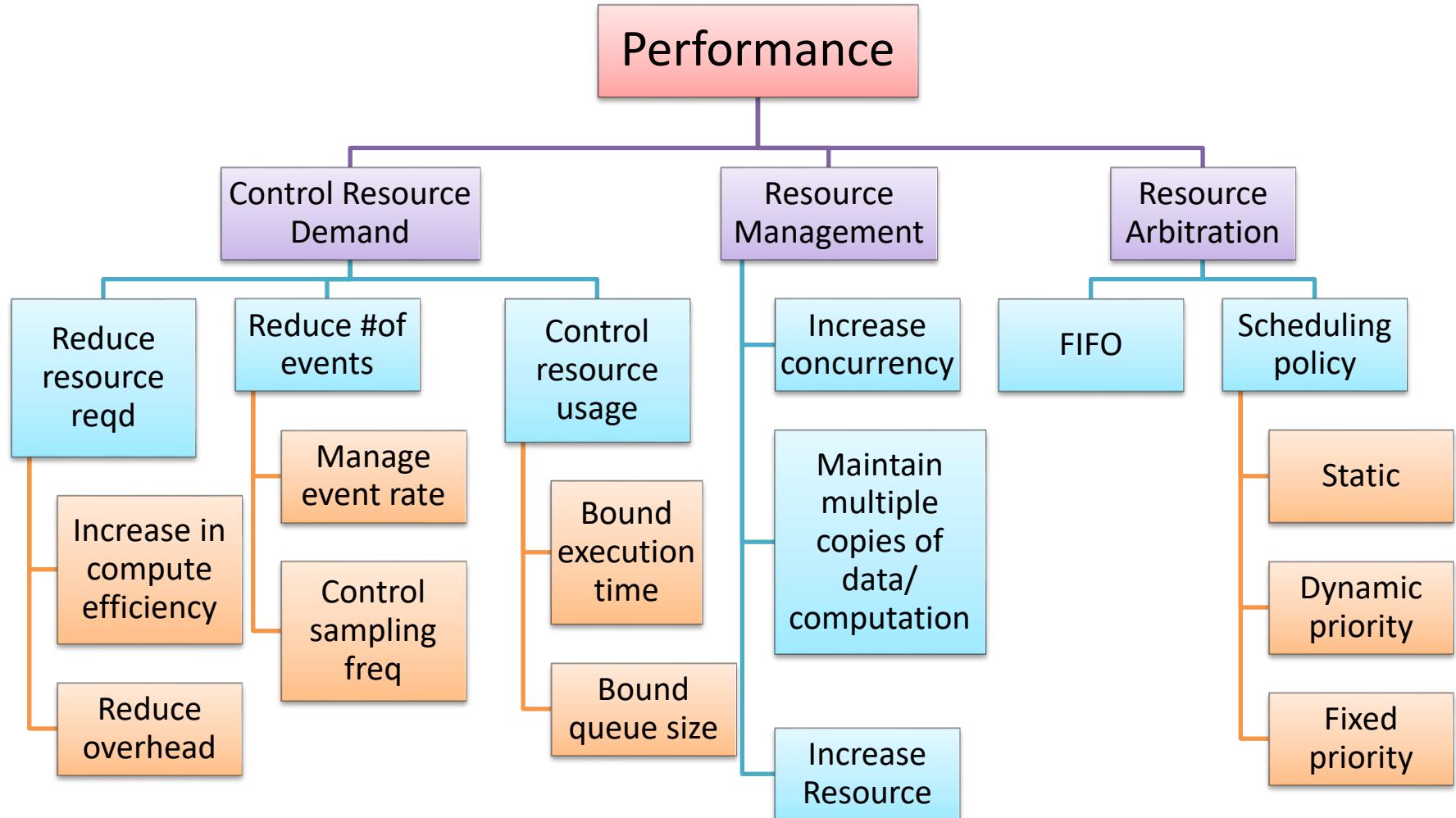
Performance Scenarios



Events and Responses

- Periodic- comes at regular intervals (real time systems)
- Stochastic- comes randomly following a probability distribution (eCommerce website)
- Sporadic- keyboard event from human
- Latency- time between arrival of stimulus and system response
- Throughput- number of txn processed/unit of time
- Jitter- allowable variation in latency
- #events not processed

Performance Tactics



Why System fails to Respond?

- Resource Consumption
 - CPU, memory, data store, network communication
 - A buffer may be sequentially accessed in a critical section
 - There may be a workflow of tasks one of which may be choked with request
- Blocking of computation time
 - Resource contention
 - Availability of a resource
 - Deadlock due to dependency of resource

Control Resource Demand

- Increase Computation Efficiency: Improving the algorithms used in performance critical areas
- Reduce Overhead
 - Reduce resource consumption when not needed
 - Use of local objects instead of RMI calls
 - Local interface in EJB 3.0
 - Remove intermediaries (conflicts with modifiability)
- Manage
 - event rate: If you have control, don't sample too many events (e.g. sampling environmental data)
 - sampling time: If you don't have control, sample them at a lower speed, leading to loss of request
- Bound
 - Execution: Decide how much time should be given on an event. E.g. iteration bound on a data-dependent algorithm
 - Queue size: Controls maximum number of queued arrivals

Manage Resources

- Increase Resources(infrastructure)
 - Faster processors, additional processors, additional memory, and faster networks
- Increase Concurrency
 - If possible, process requests in parallel
 - Process different streams of events on different threads
 - Create additional threads to process different sets of activities
- Multiple copies
 - Computations : so that it can be performed faster (client-server), MapReduce computation
 - Data:
 - use of cache for faster access and reduce contention
 - Hadoop maintains data copies to avoid data-transfer and improve data locality

Resource Arbitration

- Resources are scheduled to reduce contention
 - Processors, buffer, network
 - Architect needs to choose the right scheduling strategy
- FIFO
- Fixed Priority
 - Semantic importance
 - Domain specific logic such as request from a privileged class gets higher priority
 - Deadline monotonic (shortest job first)
- Dynamic priority
 - Round robin
 - Earliest deadline first- the job which has earliest deadline to complete
- Static scheduling
 - Also pre-emptive scheduling policy

Design Checklist for a Quality Attribute

- Allocate responsibility
 - Modules can take care of the required quality requirement
- Manage Data
 - Identify the portion of the data that needs to be managed for this quality attribute
 - Plan for various data design w.r.t. the quality attribute
- Resource Management Planning
 - How infrastructure should be monitored, tuned, deployed to address the quality concern
- Manage Coordination
 - Plan how system elements communicate and coordinate
- Binding

Performance- Design Checklist-



Allocate responsibilities

- Identify which features may involve or cause
 - Heavy workload
 - Time-critical response
- Identify which part of the system that's heavily used
- For these, analyze the scenarios that can result in performance bottleneck
- Furthermore--
 - Assign Responsibilities related to threads of control —allocation and de-allocation of threads, maintaining thread pools, and so forth
 - Assign responsibilities that will schedule shared resources or appropriately select, manage performance-related artifacts such as queues, buffers, and caches

Performance- Design Checklist-

Manage Data

- Identify the data that's involved in time critical response requirements, heavily used, massive size that needs to be loaded etc. For those data determine
 - whether maintaining multiple copies of key data would benefit performance
 - partitioning data would benefit performance
 - whether reducing the processing requirements for the creation, initialization, persistence, manipulation, translation, or destruction of the enumerated data abstractions is possible
 - whether adding resources to reduce bottlenecks for the creation, initialization, persistence, manipulation, translation, or destruction of the enumerated data abstractions is feasible.

Performance- Design Checklist-

Manage Coordination

- Look for the possibility of introducing concurrency (and obviously pay attention to thread-safety), event prioritization, or scheduling strategy
 - Will this strategy have a significant positive effect on performance? Check
 - Determine whether the choice of threads of control and their associated responsibilities introduces bottlenecks
- Consider appropriate mechanisms for example
 - stateful, stateless, synchronous, asynchronous, guaranteed delivery

Performance Design Checklist-

Resource Management

- Determine which resources (CPU, memory) in your system are critical for performance.
 - Ensure they will be monitored and managed under normal and overloaded system operation.
- Plan for mitigating actions early, for instance
 - Where heavy network loading will occur, determine whether co-locating some components will reduce loading and improve overall efficiency.
 - Ensure that components with heavy computation requirements are assigned to processors with the most processing capacity.
- Prioritization of resources and access to resources
 - scheduling and locking strategies
- Deploying additional resources on demand to meet increased loads
 - Typically possible in a Cloud and virtualized scenario

Performance Design checklist-

Binding



- For each element that will be bound after compile time, determine the
 - time necessary to complete the binding
 - additional overhead introduced by using the late binding mechanism
 - Ensure that these values do not pose unacceptable performance penalties on the system.
-

Technology choice

- Choice of technology is often governed by the organization mandate (enterprise architecture)
- Find out if the chosen technology will let you set and meet real time deadlines?
 - Do you know its characteristics under load and its limits?
- Does your choice of technology give you the ability to set
 - scheduling policy
 - Priorities
 - policies for reducing demand
 - allocation of portions of the technology to processors
- Does your choice of technology introduce excessive overhead?

Thank You



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 6.1): Software Architecture

Security and Its Tactics

Instructor: Prof. Santonu Sarkar

What is Security

A measure of the system's ability to resist unauthorized usage while still providing its services to legitimate users

- Ability to protect data and information from unauthorized access

An attempt to breach this is an “Attack”

- Unauthorized attempt to access, modify, delete data
 - Theft of money by e-transfer, modification records and files, reading and copying sensitive data like credit card number
- Deny service to legitimate users

Important aspects of Security

Security comprises of

Confidentiality

- prevention of the unauthorized disclosure of information. E.g. Nobody except you should be able to access your income tax returns on an online tax-filing site

Integrity

- prevention of the unauthorized modification or deletion of information. E.g. your grade has not been changed since your instructor assigned it

Availability

- prevention of the unauthorized withholding of information – e.g. DoS attack should not prevent you from booking railway ticket



Important aspects of

Security

Non repudiation:: An activity (say a transaction) can't be denied by any of the parties involved. E.g. you cannot deny ordering something from the Internet, or the merchant cannot disclaim getting your order.

Assurance:: Parties in an activity are assured to be who they purport to be. Typically done through authentication. E.g. if you get an email purporting to come from a bank, it is indeed from a bank.

Auditing:: System tracks activities so that it can be reconstructed later

Authorization grants a user the privileges to perform a task. For example, an online banking system authorizes a legitimate user to access his account.

Security Scenario

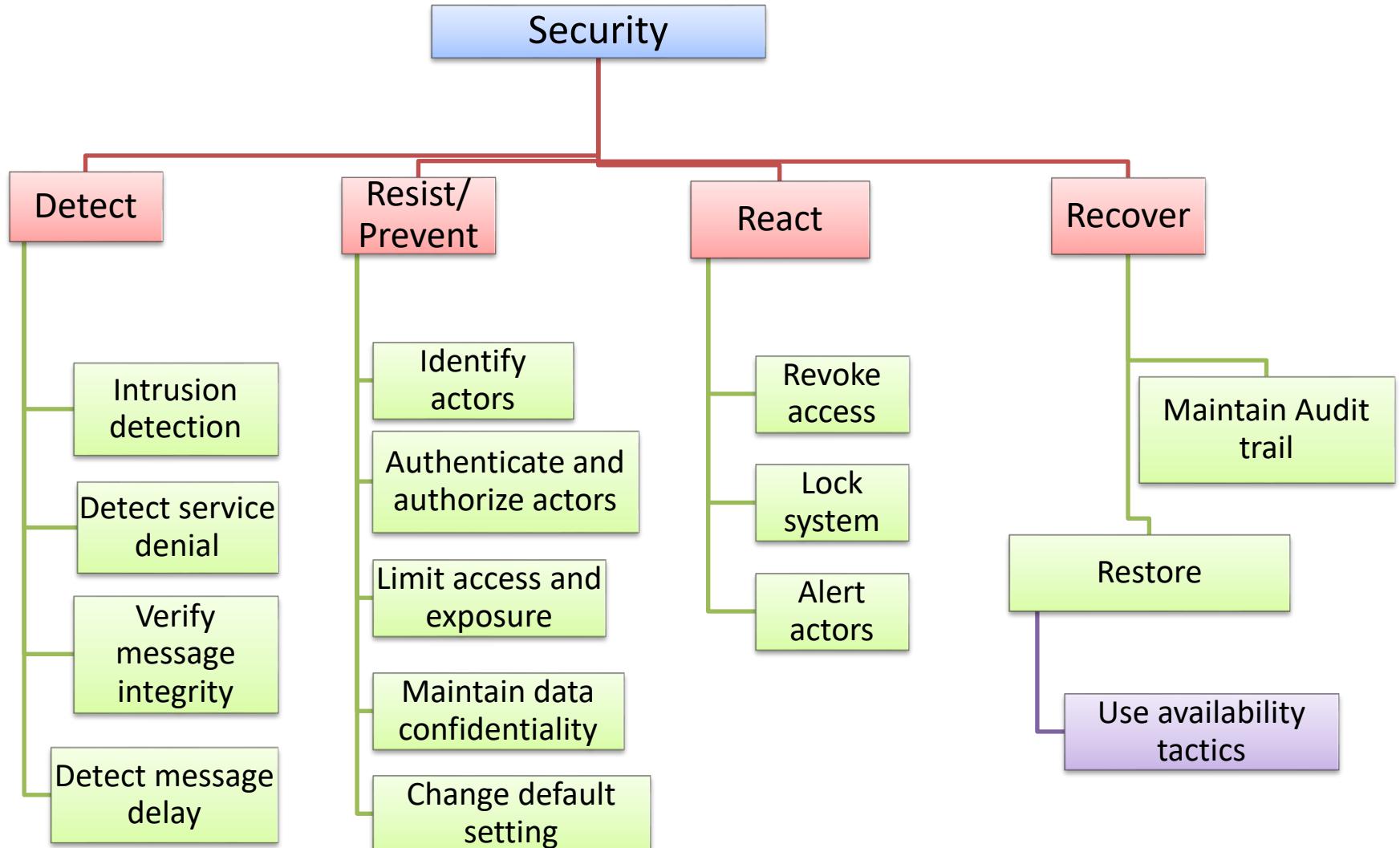
<u>WHO</u>	<u>STIMULUS</u>	<u>IMPACTED PART</u>	<u>MITIGATING ACTION</u>	<u>MEASURABLE RESPONSE</u>
An insider • Individual or a system ✓ Correctly identified OR ✓ Incorrectly identified ✓ OR unknown • Who is ✓ Internal or external ✓ Authorized or unauthorized • Has access to ✓ Limited resource ✓ Vast resource	Updates payment details • Tries to ✓ Read data ✓ Change/delete data ✓ Access system services ✓ Reduce availability of services	Pay database table System services, data • Online/offline • Within firewall or Open	<ul style="list-style-type: none"> • Authentication <ul style="list-style-type: none"> ✓ Authenticates ✓ Hides identity • Access control to data or services <ul style="list-style-type: none"> ✓ Blocks access ✓ Grants/withdraws permission to access ✓ Audits access/modification attempts • Corrective Actions <ul style="list-style-type: none"> ✓ Encrypts data ✓ Detect anomalous situation (high access req.) and informs people/another system ✓ Restricts availability 	<ul style="list-style-type: none"> Time required to circumvent security measure with Probability of success Probability of detecting attack Probability of detecting the individual responsible % of services available during attack Time to restore data/services Extent of damage-how much data is vulnerable No of access denials

Security Tactics- Close to Physical Security



- Detection:
 - Limit the access through security checkpoints
 - Enforces everyone to wear badges or checks legitimate visitors
 - Resist
 - Armed guards
 - React
 - Lock the door automatically
 - Recover
 - Keep backup of the data in a different place
-

Security Tactics



Detect Attacks

- Detect Intrusion: compare network traffic or service request patterns *within* a system to
 - a set of signatures or
 - known patterns of malicious behavior stored in a database.
 - Detect Service Denial
 - Compare the pattern or signature of network traffic *coming into* a system to historic profiles of known Denial of Service (DoS) attacks.
 - Verify Message Integrity
 - Use checksums or hash values to verify the integrity of messages, resource files, deployment files, and configuration files.
 - Detect Message Delay:
 - checking the time that it takes to deliver a message, it is possible to detect suspicious timing behavior.
-

Resist Attacks

- Identify Actors: identify the source of any external input to the system.
- Authenticate & Authorize Actors:
 - Use strong passwords, OTP, digital certificates, biometric identity
 - Use access control pattern, define proper user class, user group, role based access
- Limit Access
 - Restrict access based on message source or destination ports
 - Use of DMZ

Resist Attacks

- Limit Exposure: minimize the attack surface of a system by allocating limited number of services to each hosts
- Data confidentiality:
 - Use encryption to encrypt data in database
 - User encryption based communication such as SSL for web based transaction
 - Use Virtual private network to communicate between two trusted machines
- Separate Entities: can be done through physical separation on different servers attached to different networks, the use of virtual machines, or an “air gap”.
- Change Default Settings: Force the user to change settings assigned by default.

React to Attacks

- Revoke Access: limit access to sensitive resources, even for normally legitimate users and uses, if an attack is suspected.
- Lock Computer: limit access to a resource if there are repeated failed attempts to access it.
- Inform Actors: notify operators, other personnel, or cooperating systems when an attack is suspected or detected.

Recover From Attacks

- In addition to the Availability tactics for recovery of failed resources there is Audit.
- Audit: keep a record of user and system actions and their effects, to help trace the actions of, and to identify, an attacker.

Design Checklist- Allocation of Responsibilities

- Identify the services that needs to be secured
 - Identify the modules, subsystems offering these services
- For each such service
 - Identify actors which can access this service, and implement authentication and level of authorization for those
 - verify checksums and hash values
 - Allow/deny data associated with this service for these actors
 - record attempts to access or modify data or services
 - Encrypt data that are sensitive
 - Implement a mechanism to recognize reduced availability for this services
 - Implement notification and alert mechanism
 - Implement recover from an attack mechanism

Design Checklist- Manage Data

- Determine the sensitivity of different data fields
 - Ensure that data of different sensitivity is separated
 - Ensure that data of different sensitivity has different access rights and that access rights are checked prior to access.
 - Ensure that access to sensitive data is logged and that the log file is suitably protected.
 - Ensure that data is suitably encrypted and that keys are separated from the encrypted data.
 - Ensure that data can be restored if it is inappropriately modified.
-

Design Checklist- Manage Coordination

- For inter-system communication (applied for people also)
 - Ensure that mechanisms for authenticating and authorizing the actor or system, and encrypting data for transmission across the connection are in place.
- Monitor communication
 - Monitor anomalous communication such as
 - unexpectedly high demands for resources or services
 - Unusual access pattern
 - Mechanisms for restricting or terminating the connection.

Design Checklist- Manage Resource

- Define appropriate grant or denial of resources
- Record access attempts to resources
- Encrypt data
- Monitor resource utilization
 - Log
 - Identify suddenly high demand to a particular resource-for instance high CPU utilization at an unusual time
- Ensure that a contaminated element can be prevented from contaminating other elements.
- Ensure that shared resources are not used for passing sensitive data from an actor with access rights to that data to an actor without access rights.
 - .

Design checklist- Binding

- Runtime binding of components can be untrusted. Determine the following
 - Based on situation implement certificate based authentication for a component
 - Implement certification management, validation
 - Define access rules for components that are dynamically bound
 - Implement audit trail for whenever a late bound component tries to access records
 - System data should be encrypted where the keys are intentionally withheld for late bound components

Design Checklist- Technology choice

Choice of technology is often governed by the organization mandate (enterprise architecture)

- Decide tactics first. Based on the tactics, ensure that your chosen technologies support the tactics
- Determine what technology are available to help user authentication, data access rights, resource protection, data encryption
- Identify technology and tools for monitoring and alert



SS ZG653 (RL 6.2): Software Architecture

Testability and Its Tactics

Instructor: Prof. Santonu Sarkar



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

What is Testability

The ease with which software can be made to demonstrate its faults through testing

If a fault is present in a system, then we want it to fail during testing as quickly as possible.

At least 40% effort goes for testing

- Done by developers, testers, and verifiers (tools)

Specialized software for testing

- Test harness
- Simple playback capability
- Specialized testing chamber

Testable Software

Dijkstra's Thesis

- Test can't guarantee the absence of errors, but it can only show their presence.
- Fault discovery is a probability
 - That the next test execution will fail and exhibit the fault
- A perfectly testable code – each component's internal state must be controllable through inputs and output must be observable
 - Error-free software does not exist.

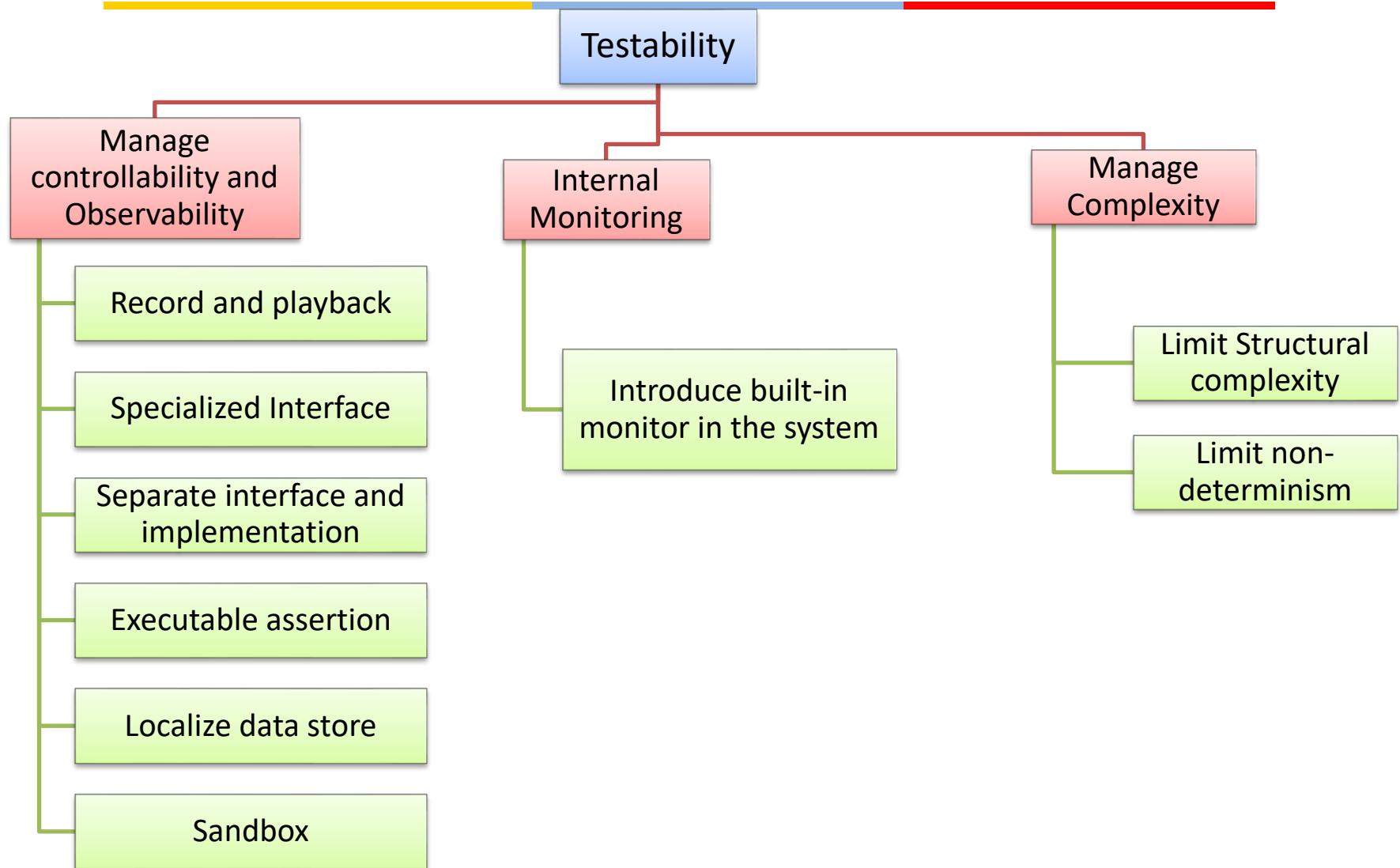
Testability Scenario

	<u>WHO</u>	<u>STIMULUS</u>	<u>IMPACTED PART</u>	<u>RESPONSE ACTION</u>	<u>MEASURABLE RESPONSE</u>
A unit tester	<ul style="list-style-type: none"> • Unit tester (typically unit developers) • Integration tester • System tester or client acceptance team • System user 	<p>Milestone in the development process is met</p> <ul style="list-style-type: none"> ✓ Completion of design ✓ Completion of coding ✓ Completion of integration 	<p>Component or whole system</p> <ul style="list-style-type: none"> • Design time • Development time • Compile time • Integration time 	<ul style="list-style-type: none"> • Prepare test environment • Access state values • Access computed values 	<ul style="list-style-type: none"> • %executable statements executed (code coverage) • Time to test • Time to prepare test environment • Length of longest dependency chain in test • Probability of failure if fault exists
	Performs unit test	Component that has controllable interface After component is complete	Observe the output for inputs provided	Coverage of 85% is achieved in 2 hours	

Goal of Testability Tactics

- Using testability tactics the architect should aim to reduce the high cost of testing when the software is modified
- Two categories of tactics
 - Introducing controllability and observability to the system during design
 - The second deals with limiting complexity in the system's design

Testability Tactics



Control and Observe System State

- Specialized Interfaces for testing:
 - to control or capture variable values for a component either through a test harness or through normal execution.
 - Use a special interface that a test harness can use
 - Make use of some metadata through this special interface
 - Record/Playback: capturing information crossing an interface and using it as input for further testing.
 - Localize State Storage: To start a system, subsystem, or module in an arbitrary state for a test, it is most convenient if that state is stored in a single place.
-

Control and Observe System State

- Interface and implementation
 - If they are separated, implementation can be replaced by a stub for testing rest of the system
- Sandbox: isolate the system from the real world to enable experimentation that is unconstrained by the worry about having to undo the consequences of the experiment.
- Executable Assertions: assertions are (usually) hand coded and placed at desired locations to indicate when and where a program is in a faulty state.

Manage Complexity

- Limit Structural Complexity:
 - avoiding or resolving cyclic dependencies between components,
 - isolating and encapsulating dependencies on the external environment
 - reducing dependencies between components in general.
 - Limit Non-determinism: finding all the sources of non-determinism, such as unconstrained parallelism, and remove them out as far as possible.
-

Internal Monitoring

- Implement a built-in monitoring mechanism
 - One should be able to turn on or off
 - one example is logging
 - Performed typically by instrumentation- AOP, Preprocessor macro. Instrument the code to introduce recorder at some point

Design Checklist- Allocation of Responsibility

Identify the services are most critical and hence need to be most thoroughly tested.

- Identify the modules, subsystems offering these services
- For each such service
 - Ensure that internal monitoring mechanism like logging is well designed
 - Make sure that the allocation of functionality provides
 - low coupling,
 - strong separation of concerns, and
 - low structural complexity.

Design Checklist- Testing Data

- Identify the data entities that are related to the critical services need to be most thoroughly tested.
 - Ensure that creation, initialization, persistence, manipulation, translation, and destruction of these data entities are possible--
 - State Snapshot: Ensure that the values of these data entities can be captured if required, while the system is in execution or at fault
 - Replay: Ensure that the desired values of these data entities can be set (state injection) during testing so that it is possible to recreate the faulty behavior
-

Design Checklist- Testing Infrastructure

- . Is it possible to inject faults into the communication channel and monitoring the state of the communication
- . Is it possible to execute test suites and capture results for a distributed set of systems?
- . Testing for potential race condition- check if it is possible to explicitly map
 - . processes to processors
 - . threads to processes

So that the desired test response is achieved and potential race conditions identified

Design Checklist- Testing resource binding

- Ensure that components that are bound later than compile time can be tested in the late bound context
 - E.g. loading a driver on-demand
- Ensure that late bindings can be captured in the event of a failure, so that you can re-create the system's state leading to the failure.
- Ensure that the full range of binding possibilities can be tested.

Design Checklist- Resource Management

- Ensure there are sufficient resources available to execute a test suite and capture the results
 - Ensure that your test environment is representative of the environment in which the system will run
 - Ensure that the system provides the means to:
 - test resource limits
 - capture detailed resource usage for analysis in the event of a failure
 - inject new resources limits into the system for the purposes of testing
 - provide virtualized resources for testing
-

Choice of Tools

- Determine what tools are available to help achieve the testability scenarios
 - Do you have regression testing, fault injection, recording and playback supports from the testing tools?
- Does your choice of tools support the type of testing you intend to carry on?
 - You may want a fault-injection but you need to have a tool that can support the level of fault-injection you want
 - Does it support capturing and injecting the data-state



SS ZG653 (RL 6.3): Software

Architecture

Interoperability and Its Tactics

Instructor: Prof. Santonu Sarkar



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

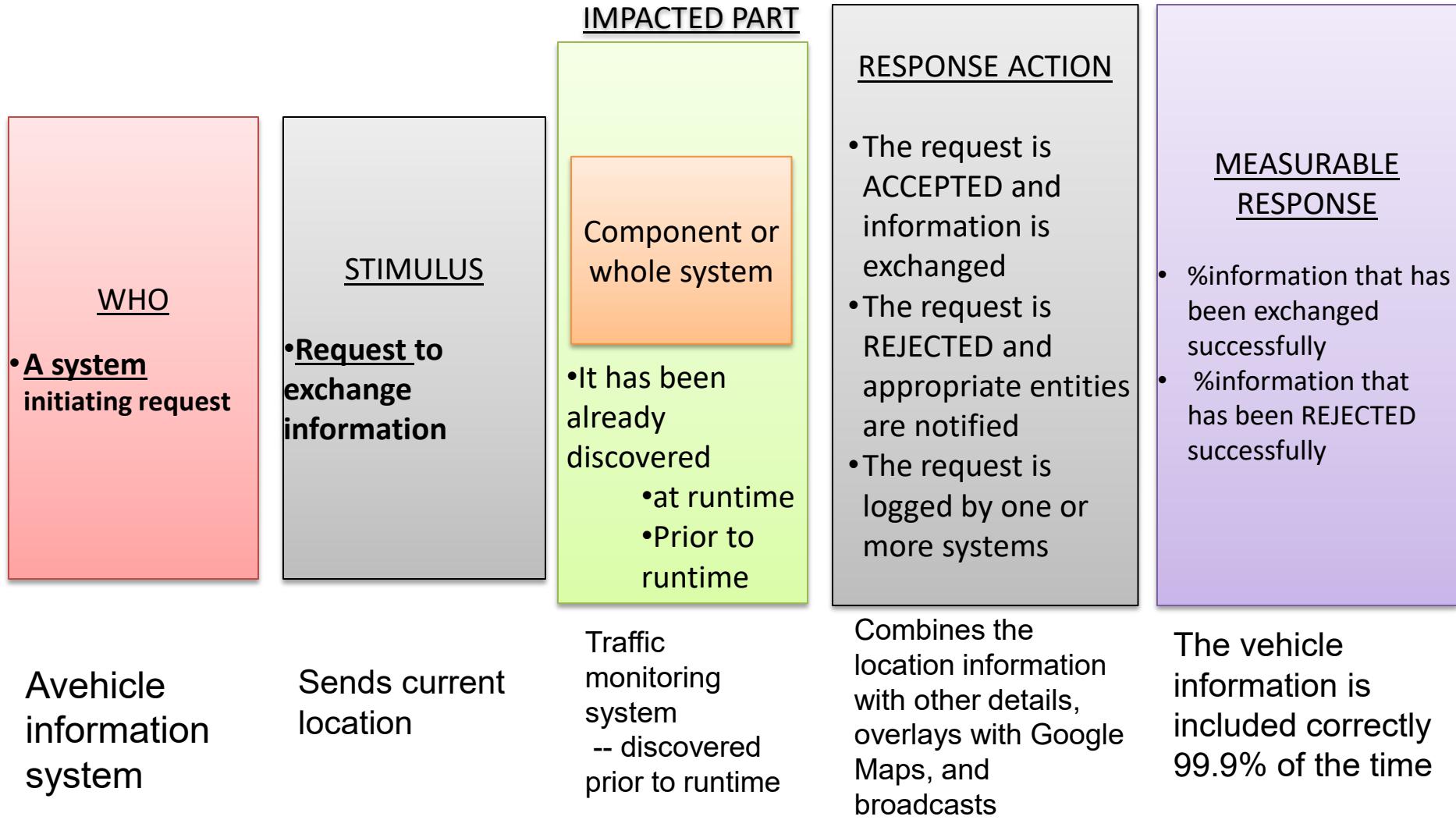
Interoperability

- Ability that two systems can usefully exchange information through an interface
 - Ability to transfer data (syntactic) and interpret data (semantic)
 - Information exchange can be direct or indirect
 - Interface
 - Beyond API
 - Need to have a set of assumptions you can safely make about the entity exposing the API
 - Example- you want to integrate with Google Maps
-

Why Interoperate?

- The service provided by Google Maps are used by unknown systems
 - They must be able to use Google Maps w/o Google knowing who they can be
- You may want to construct capability from variety of systems
 - A traffic sensing system can receive stream of data from individual vehicles
 - Raw data needs to be processed
 - Need to be fused with other data from different sources
 - Need to decide the traffic congestion
 - Overlay with Google Maps

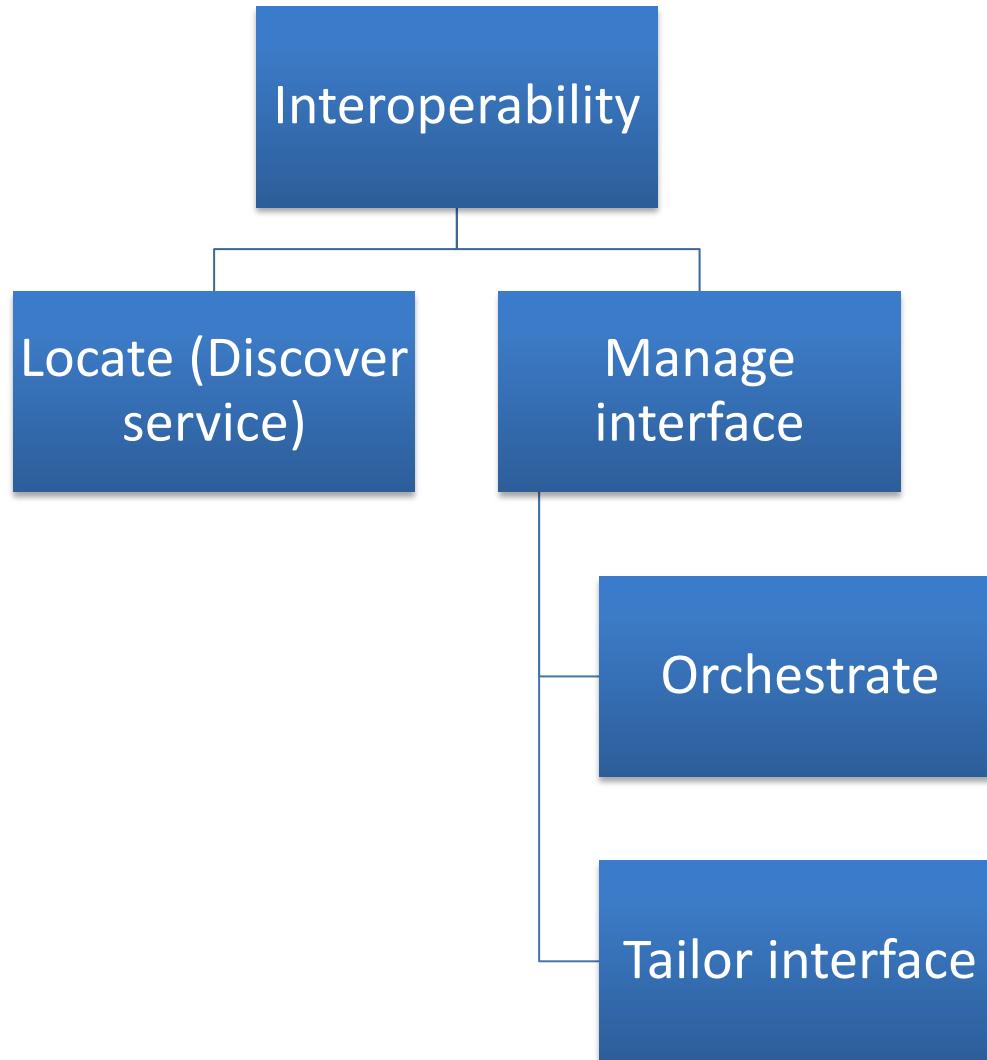
Interoperability Scenario



Notion of Interface

- Information exchange
 - Can be as simple as A calling B
 - A and B can exchange implicitly w/o direct communication
 - Operation Desert Storm 1991: Anti-missile system failed to exchange information (intercept) an incoming ballistic rocket
 - The system required periodic restart in order to recalibrate its position. Since it wasn't restarted, the information wasn't correctly captured due to error accumulation
- Interface
 - Here it also means that a set of assumptions that can be made safely about this entity
 - E.g. it is safe to assume that the API of anti-missile system DOES NOT give information about gradual degradation

Tactics



Interoperability Tactics

- Locate (Discover service)
 - Identify the service through a known directory service. Here service implies a set of capabilities available through an interface
 - By name, location or other attributes

Interoperability Tactics

Manage interface

- Orchestrate
 - Co-ordinate and manage a sequence of services.
Example- workflow engines containing scripts of interaction
 - Mediator design pattern for simple orchestration. BPEL language for complex orchestration
- Tailor interface
 - Add or remove capability from an interface (hide a particular function from an untrusted user)
 - Use Decorator design pattern for this purpose

REpresentational State Transfer (REST)

REST is an architectural pattern where services are described using an uniform interface. RESTful services are viewed as a hypermedia resource. REST is stateless.

REST Verb	CRUD Operation	Description
POST	CREATE	Create a new resource.
GET	RETRIEVE	Retrieve a representation of the resource.
PUT	UPDATE	Update a resource.
DELETE	DELETE	Delete a resource.

Google Suggest : <http://suggestqueries.google.com/complete/search?output=toolbar&hl=en&q=satyajit%20ray>

Yahoo Search:

<http://search.yahooapis.com/WebSearchService/V1/webSearch?appid=YahooDemo&query=accenture>

REST vs. SOAP/WSDL

- Simply put, the community has claimed that SOAP and WSDL have become too grandiose and comprehensive to achieve the “agility” touted by SOA (Seeley, R., “Burton sees the future of SOA and it is REST,” SearchWebService.com, May 30, 2007)

	SOAP/WSDL	REST
Purpose	Message exchange between two applications/systems	Access and manipulating a hypermedia system
Origin	RPC	WWW
Functionality	Rich	Minimal
Interaction	Orchestrated event-based	Client/server (request/response)
Focus	Process-oriented	Data-oriented
Methods/operations	Varies depending on the service	Fixed
Reuse	Centrally governed	Little/no governance (focus on ease of use instead)
Interaction context	Can be maintained in both client and server	Only on client
Format	SOAP in, SOAP out	URI (+POX) in, POX out
Transport	Transport independent	HTTP only
Security	WS-Security	HTTP authentication + SSL

Design Checklist- Interoperability

- Allocation of Responsibilities: Check which system features need to interoperate with others. For each of these features, ensure that the designers implement
 - Accepting and rejecting of requests
 - Logging of request
 - Notification mechanism
 - Exchange of information
- Coordination Model: Coordination should ensure performance SLAs to be met. Plan for
 - Handling the volume of requests
 - Timeliness to respond and send the message
 - Currency of the messages sent
 - Handle jitters in message arrival times

Design Checklist-Interoperability

Data Model

- Identify the data to be exchanged among interoperating systems
- If the data can't be exchanged due to confidentiality, plan for data transformation before exchange

Identification of Architectural Component

- The components that are going to interoperate should be available, secure, meet performance SLA (consider design-checklists for these quality attributes)

Design Checklist- Interoperability

Resource Management

- Ensure that system resources are not exhausted (flood of request shouldn't deny a legitimate user)
- Consider communication load
- When resources are to be shared, plan for an arbitration policy

Binding Time

- Ensure that it has the capability to bind unknown systems
- Ensure the proper acceptance and rejection of requests
- Ensure service discovery when you want to allow late binding

Technology Choice

- Consider technology that supports interoperability (e.g. web-services)

Thank You



Reference Chapter 16
Software Architecture in Practice
Third Edition
Len Bass
Paul Clements
Rick Kazman

Software Architecture

Designing & Documenting the Architecture #1

Architecture and Requirements



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal
Module 19 (RL6.1)

Architecture and Requirements

- Define Architecturally Significant Requirements (ASR)
- Gathering (ASR) from Requirement document, Business goals & Stakeholders
- Capturing ASR in an Utility Tree for further refinement
- Architecture Design strategy and Attribute –Driven Design Method



Define Architecturally Significant Requirements (ASR)

Requirements

- Architectures exist to build systems that satisfy requirements.
- But, to an architect, not all requirements are created equal.
- An *architecturally significant requirement* (ASR) is a requirement that will have a profound effect on the architecture.
- How do we find those?

ASRs and Requirements Documents



- An obvious location to look for candidate ASRs is in the requirements documents or in user stories.
- Requirements should be in requirements documents!
- Unfortunately, this is not usually the case.
- Why?

Don't Get Your Hopes Up

- Many projects don't create or maintain the detailed, high-quality requirements documents.
- Standard requirements pay more attention to functionality than quality attributes.
- Most of what is in a requirements specification does not affect the architecture.
- No architect just sits and waits until the requirements are “finished” before starting work. The architect *must* begin while the requirements are still in flux.

When does the Architect Start?

Don't Get Your Hopes Up

- Quality attributes, when captured at all, are often captured poorly.
 - “The system shall be modular”
 - “The system shall exhibit high usability”
 - “The system shall meet users’ performance expectations”
- Much of what is useful to an architect is not in even the best requirements document.
 - ASRs often derive from business goals in the development organization itself
 - Developmental qualities (such as teaming) are also out of scope

Most ASRs are *not obvious*.



Gathering (ASR) from Requirement document, Business goals & Stakeholders

Sniffing Out ASRs

Design Decision Category

Allocation of Responsibilities

Coordination Model

Data Model

Management of Resources

Mapping among Architectural Elements

Binding Time Decisions

Choice of Technology

Look for Requirements Addressing ...

Planned evolution of responsibilities, user roles, system modes, major processing steps, commercial packages

Properties of the coordination (timeliness, currency, completeness, correctness, and consistency)

Names of external elements, protocols, sensors or actuators (devices), middleware, network configurations (including their security properties)

Evolution requirements on the list above

Processing steps, information flows, major domain entities, access rights, persistence, evolution requirements

Time, concurrency, memory footprint, scheduling, multiple users, multiple activities, devices, energy usage, soft resources (buffers, queues, etc.)

Scalability requirements on the list above

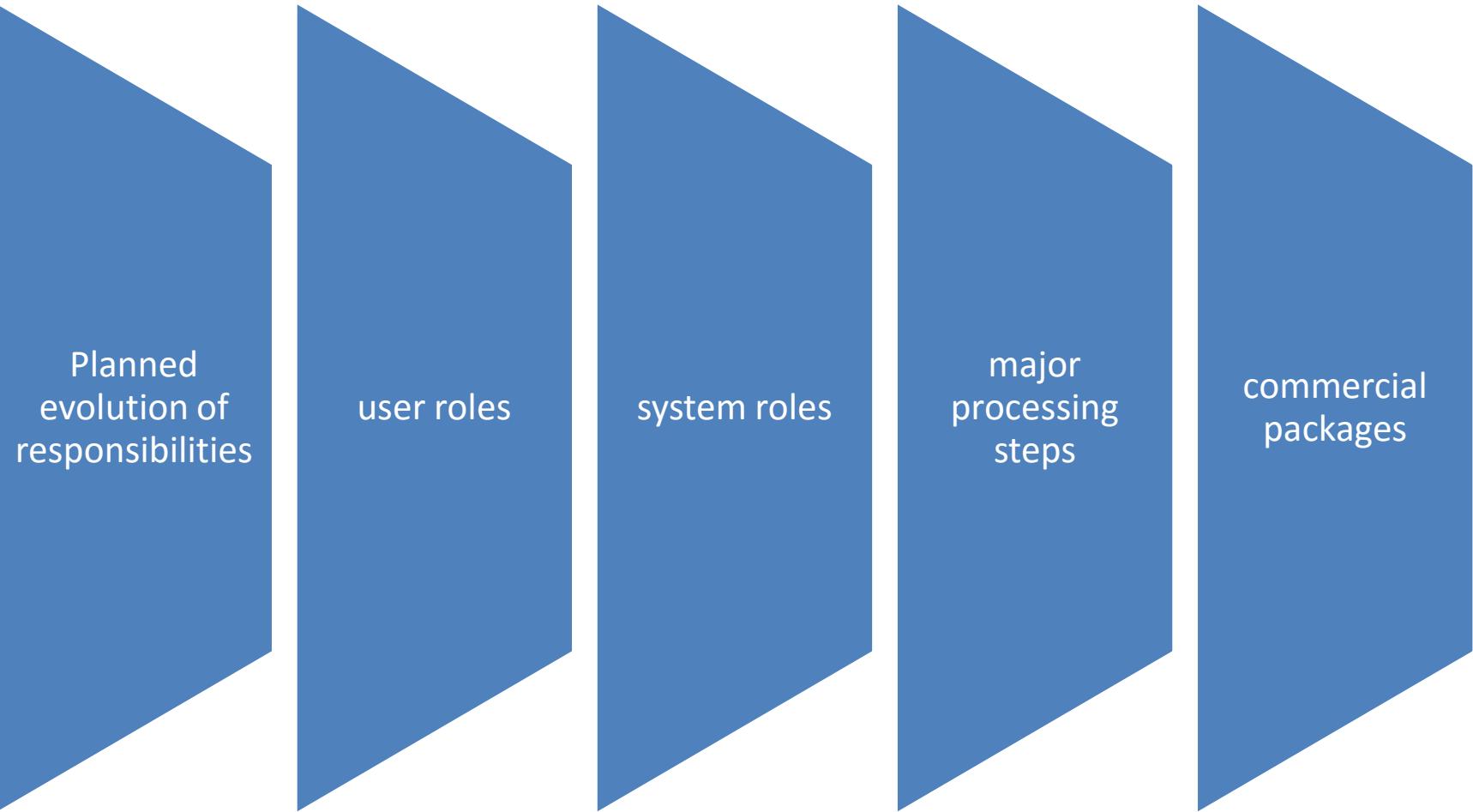
Plans for teaming, processors, families of processors, evolution of processors, network configurations

Extension of or flexibility of functionality, regional distinctions, language distinctions, portability, calibrations, configurations

Named technologies, changes to technologies (planned and unplanned)

Requirements that can affect:

ALLOCATION OF RESPONSIBILITY



Requirements that can affect: COORDINATION MODEL



Properties of coordination

- timeliness
- currency
- completeness
- correctness
- Consistence

Names of

- external elements
- protocols
- sensors
- actuators (devices),
- middleware.
- network configurations (including their security properties)

Evolution requirements on the list at the left

Requirements that can affect:

DATA MODEL

Processing
steps

information
flow

major
domain
entities

access rights

persistence

evolution
requirements

Requirements that can affect:

MANAGEMENT OF RESOURCES

Time

concurrency

memory footprint

scheduling

multiple users

multiple activities

devices

energy usage

soft resources (buffers, queues etc)

Scalability requirements on the list above

MAPPING AMONG ARCHITECTURAL ELEMENTS

Plans for

- teaming
- processors
- families of processors
- evolution of processors
- network configuration

Requirements that can affect: BINDING TIME DECISIONS

Extension of or
flexibility of
functionality

regional
distinctions

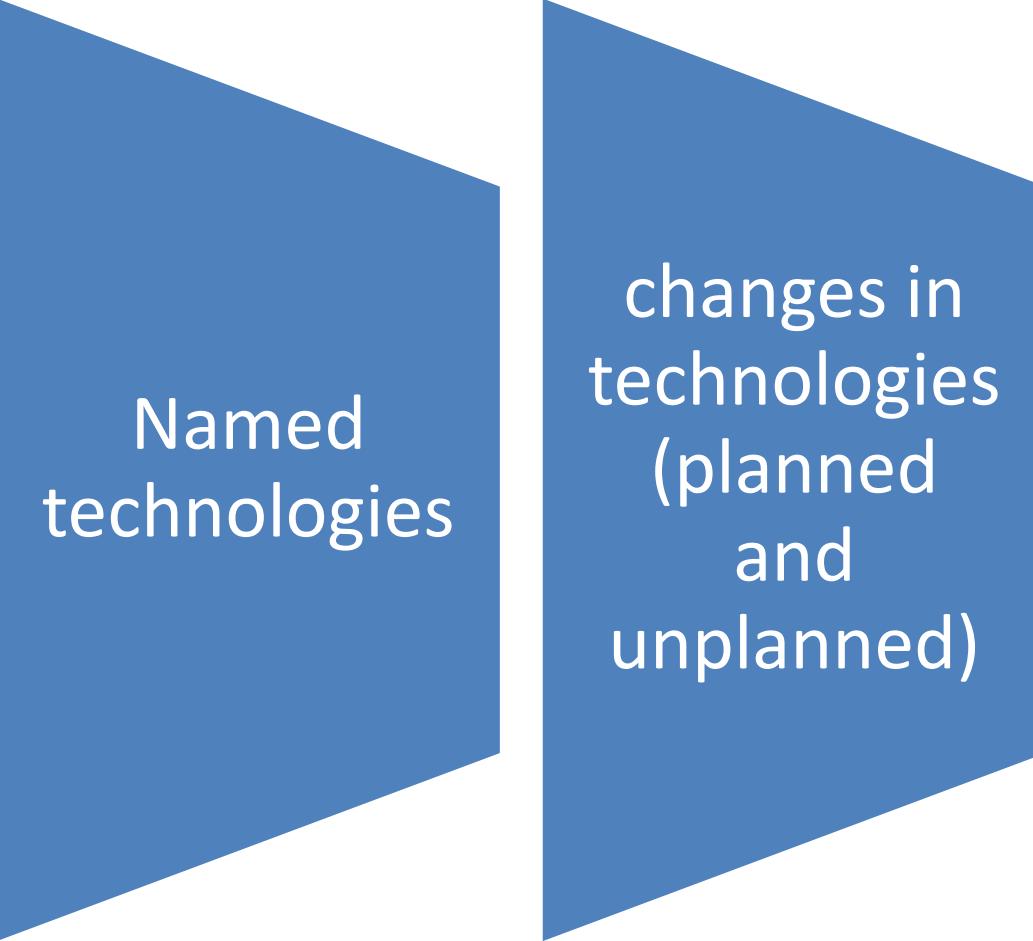
portability

calibration

configurations

Requirements that can affect:

CHOICE OF TECHNOLOGY



Named
technologies

changes in
technologies
(planned
and
unplanned)

Gathering ASRs from Stakeholders

- Say your project won't have the QAs nailed down by the time you need to start your design work.
- What do you do?
- Stakeholders often have *no idea* what QAs they want in a system
 - if you insist on quantitative QA requirements, you're likely to get numbers that are arbitrary.
 - at least some of those requirements will be very difficult to satisfy.
- Architects often have very good ideas about what QAs are reasonable to provide.
- Interviewing the relevant stakeholders is the surest way to learn what they know and need.

Gathering ASRs from Stakeholders

The results of stakeholder interviews should include

- a list of architectural drivers
- a set of QA scenarios that the stakeholders (as a group) prioritized.

This information can be used to:

- **refine** system and software **requirements**
- understand and clarify the system's **architectural drivers**
- provide rationale for why the architect subsequently made certain **design decisions**
- guide the development of **prototypes and simulations**
- influence the **order in which the architecture is developed**.

Quality Attribute Workshop

- The QAW is a facilitated, stakeholder-focused method to generate, prioritize, and refine **quality attribute scenarios** before the software architecture is completed.
- The QAW is focused on **system-level concerns** and specifically the role that software will play in the system.



The logo features the letters 'Q A W' in a bold, red, sans-serif font. The letters are slightly overlapping and have a 3D effect, appearing to float within a black rectangular frame.

QAW Steps

Step 1: QAW Presentation and Introductions.

- QAW facilitators describe the motivation for the QAW and explain each step of the method.

Step 2: Business/Mission Presentation.

- The stakeholder representing the business concerns behind the system presents the system's business context, broad functional requirements, constraints, and known quality attribute requirements.
- The quality attributes that will be refined in later steps will be derived largely from the business/mission needs presented in this step.

Step 3: Architectural Plan Presentation.

- The architect will present the system architectural plans as they stand.
- This lets stakeholders know the current architectural thinking, to the extent that it exists.

Step 4: Identification of Architectural Drivers.

- The facilitators will share their list of key architectural drivers that they assembled during Steps 2 and 3, and ask the stakeholders for clarifications, additions, deletions, and corrections.
- The idea is to reach a consensus on a distilled list of architectural drivers that includes overall requirements, business drivers, constraints, and quality attributes.

QAW Steps

Step 5: Scenario Brainstorming.

- Each stakeholder expresses a scenario representing his or her concerns with respect to the system.
- Facilitators ensure that each scenario has an explicit stimulus and response.
- The facilitators ensure that at least one representative scenario exists for each architectural driver listed in Step 4.

Step 6: Scenario Consolidation.

- Similar scenarios are consolidated where reasonable.
- Consolidation helps to prevent votes from being spread across several scenarios that are expressing the same concern.

Step 7: Scenario Prioritization.

- Prioritization of the scenarios is accomplished by allocating each stakeholder a number of votes equal to 30 percent of the total number of scenarios

Step 8: Scenario Refinement.

- The top scenarios are refined and elaborated.
- Facilitators help the stakeholders put the scenarios in the six-part scenario form of source-stimulus-artifact-environment-response-response measure.

ASRs from Business Goals

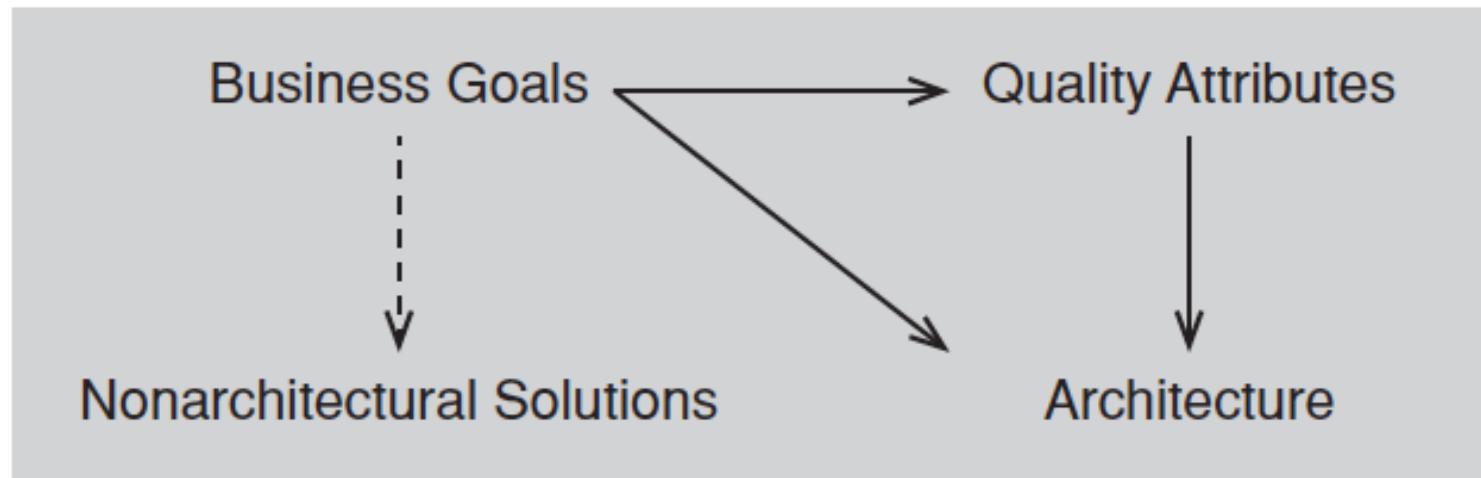
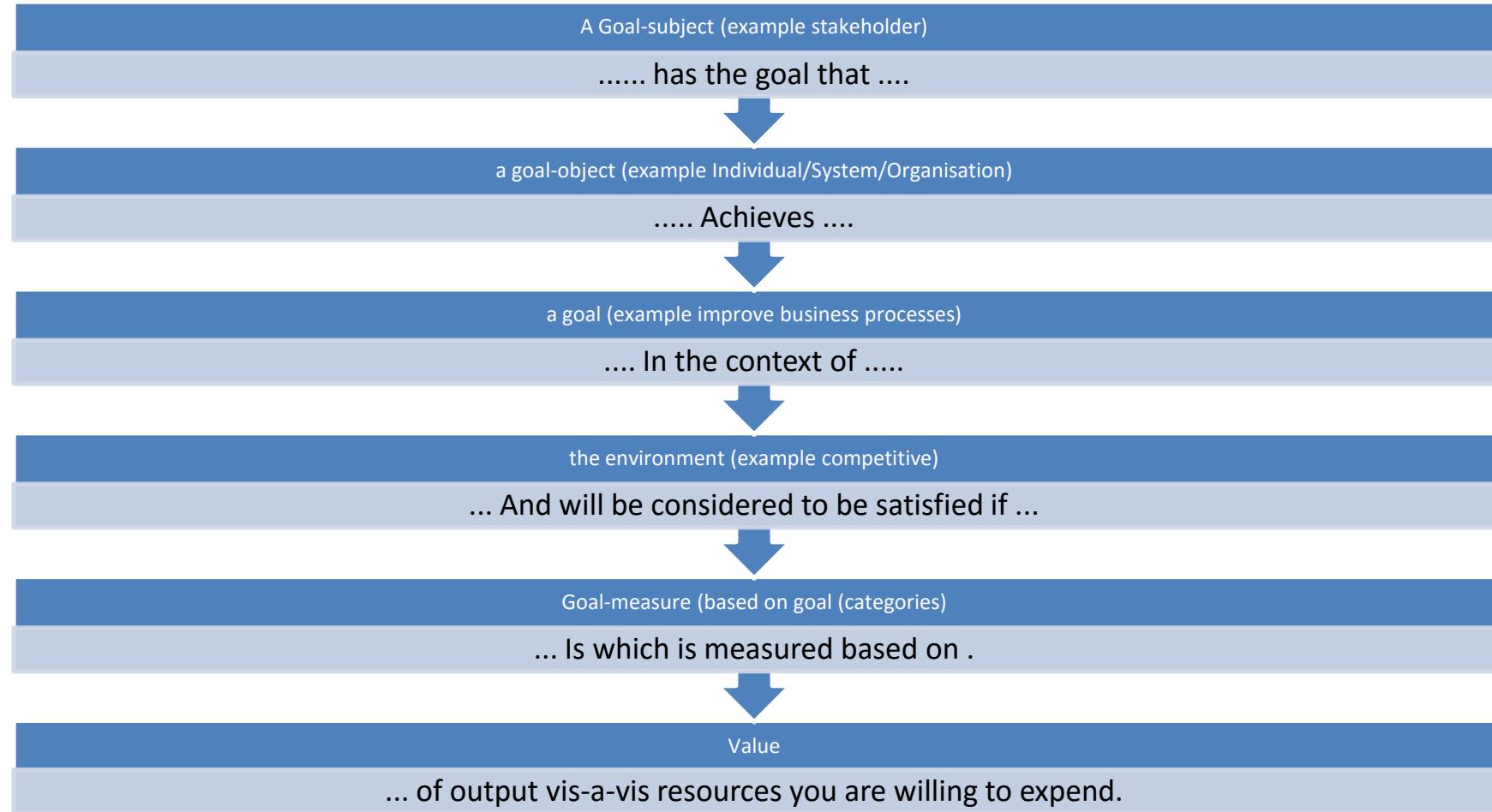


FIGURE 3.2 Some business goals may lead to quality attribute requirements (which lead to architectures), or lead directly to architectural decisions, or lead to nonarchitectural solutions.

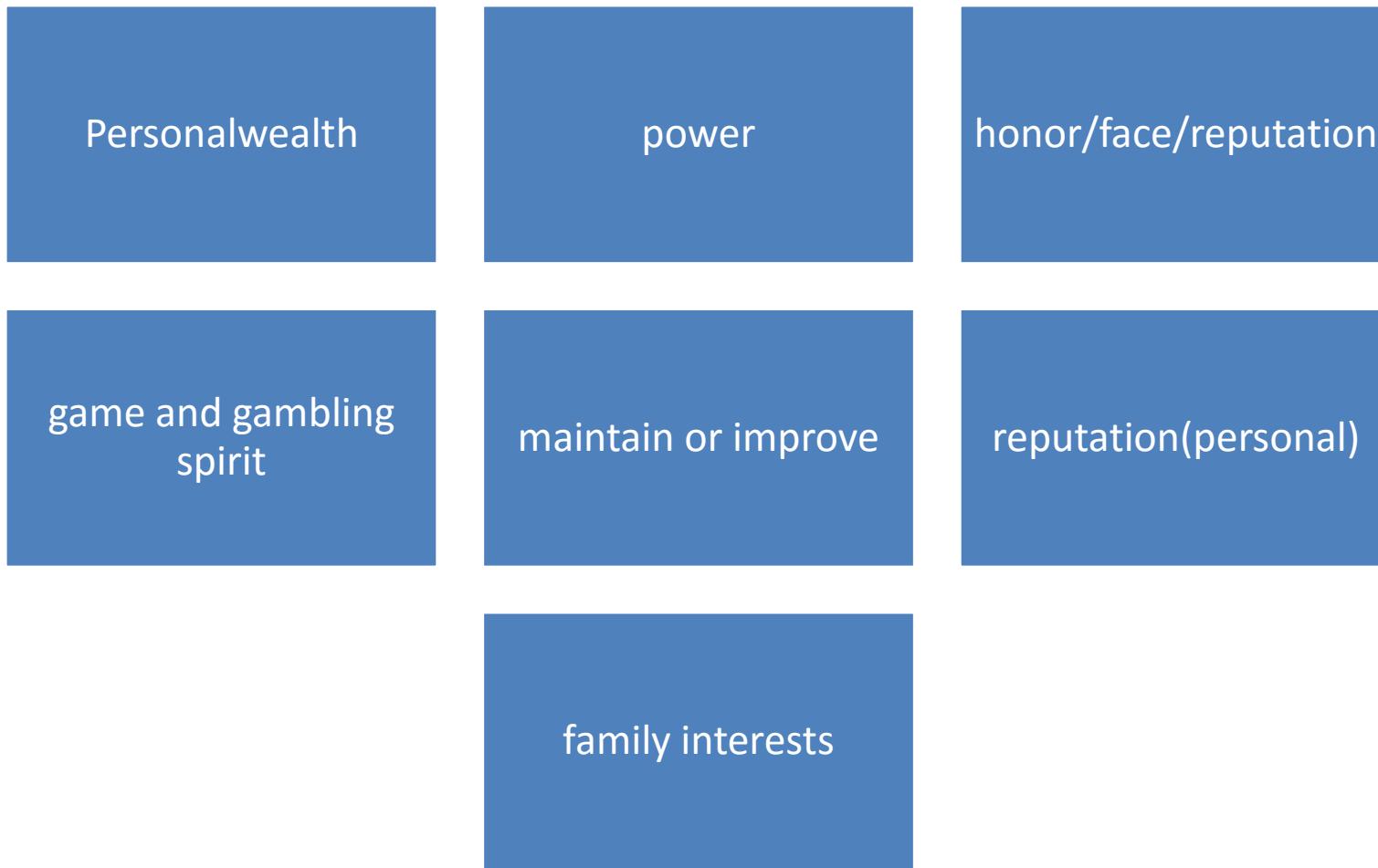
A General Scenario for Business Goals



Examples of Business Goals For the following goal-objects

- Individual
- System
- Portfolio
- Organisation's Employees
- Organisation's Shareholders
- Organisation
- Nation
- Society

Goal-Object: Individual Business Goals



Goal-Object: System

Business Goals->



Goal-Object: Portfolio

Business Goals->



Reduce cost of development	cost leadership,	differentiation,	reduce cost of	retirement
smooth transition to follow-on systems	replace legacy systems	replace labor with automation	diversify operational sequence	eliminate intermediate stages
automate tracking of business events	collect/communicate/retrieve operational knowledge	improve decision making	coordinate across distance	align task and process
manage on basis of process measurements	operate effectively within the competitive environment, the technological environment, or the customer environment	Create something new	provide the best quality products and services possible	be the leading innovator in the industry

Goal-Object: Organisation's Employees

Business Goals->



Provide high rewards and benefits to employees,

create a pleasant and friendly work place,

have satisfied employees

fulfill responsibility toward employees

maintain jobs of work force on legacy systems

Business Goals->

Maximize dividends
for the shareholders

Goal-Object: Organisation Business Goals->



Growth of the business	continuity of the business	maximize profits over the short run	maximize profits over the long run
survival of the organization	maximize the company's net assets and reserves	be a market leader	maximize the market share
expand or retain market share	enter new markets	maximize the company's rate of growth	keep tax payments to a minimum
increase sales growth	maintain or improve reputation	achieve business goals through financial objectives	run a stable organization

Goal-Object: Nation Business Goals->



Patriotism

national
pride

national
security

national
welfare

Goal-Object: Society Business Goals->



Run an ethical organization

responsibility toward society

be a socially responsible company,

be of service to the community

operate effectively within social environment

operate effectively within legal environment

Categories of Business Goals, to Aid in Elicitation

BUSINESS GOAL	• GOAL-MEASURE
Contributing to the growth and continuity of the organisation	• Time that business remains viable
Meeting Financial objectives	• Financial Performance vs. objectives
Meeting personal objectives	• Promotion or raise achieved in period
Meeting responsibilities to employees	• Employee satisfaction; turnover rate
Meeting responsibilities to society	• Contribution to trade deficit
Meeting responsibilities to state	• Stock price, dividends
Meeting responsibilities to shareholders	• Market Share
Managing market position	• Time to carry out business
Improving business processes	• Quality measures of products
Managing the quality and reputation of the products	• Technology-related problems
Managing change in environmental factors	• Time window for achievement

Expressing Business Goals

Business goal scenario, 7 parts



1. Goal-source

The people or written artifacts providing the goal.



2. Goal-subject

The stakeholders who own the goal and wish it to be true.

Each stakeholder might be an individual or the organization itself



3. Goal-object

The entities to which the goal applies.



4. Environment

The context for this goal

Environment may be social, legal, competitive, customer, and technological

Expressing Business Goals

Business goal scenario, 7 parts

5. Goal

Any business goal articulated by the goal-source.

SEE LIST IN PREVIOUS SLIDE



6. Goal-measure

A testable measurement to determine how one would know if the goal has been achieved. The goal-measure should usually include a time component, stating the time by which the goal should be achieved.



7. Pedigree and value

The degree of confidence the person who stated the goal has in it

The goal's volatility and value.

PALM: A Method for Eliciting Business Goals



PALM is a seven-step method.

Nominally carried out over a day and a half in a workshop.

Attended by architect(s) and stakeholders who can speak to the relevant business goals.

PALM Steps

PALM overview presentation

Overview of PALM, the problem it solves, its steps, and its expected outcomes.



Business drivers presentation.

Briefing of business drivers by project management

What are the goals of the customer organization for this system?

What are the goals of the development organization?



Architecture drivers presentation

Briefing by the architect on the driving business and quality attribute requirements: the ASRs.



Business goals elicitation

Business goals are elaborated and expressed as scenarios.

Consolidate almost-alike business goals to eliminate duplication.

Participants prioritize the resulting set to identify the most important goals.

PALM Steps

Identification of potential quality attributes from business goals.

For each important business goal scenario, participants describe a quality attribute that (if architected into the system) would help achieve it.
If the QA is not already a requirement, this is recorded as a finding.



Assignment of pedigree to existing quality attribute drivers.

For each architectural driver identify which business goals it is there to support.

If none, that's recorded as a finding.

Otherwise, we establish its pedigree by asking for the source of the quantitative part.



Exercise conclusion

Review of results, next steps, and participant feedback.



Capturing ASR in an Utility Tree for further refinement

Capturing ASRs in a Utility Tree

An ASR must have the following characteristics:

A profound impact on the architecture

- Including this requirement will very likely result in a different architecture than if it were not included.

A high business or mission value

- If the architecture is going to satisfy this requirement it must be of high value to important stakeholders.

Utility Tree

A way to record ASRs all in one place.

Establishes priority of each ASR in terms of

- Impact on architecture
- Business or mission value

ASRs are captured as scenarios.

Root of tree is placeholder node called “Utility”.

Second level of tree contains broad QA categories.

Third level of tree refines those categories.

Utility Tree Example (excerpt)

Quality Attribute	Attribute Refinement	ASR
Performance	Transaction response time	A user updates a patient's account in response to a change-of-address notification while the system is under peak load, and the transaction completes in less than 0.75 second. (H,M)
	Throughput	A user updates a patient's account in response to a change-of-address notification while the system is under double the peak load, and the transaction completes in less than 4 seconds. (L,M)
Usability	Proficiency training	At peak load, the system is able to complete 150 normalized transactions per second. (M,M)
	Normal operations	A new hire with two or more years' experience in the business becomes proficient in Nightingale's core functions in less than 1 week. (M,L)
Configurability	User-defined changes	A user in a particular context asks for help, and the system provides help for that context, within 3 seconds. (H,M)
	Normal operations	A hospital payment officer initiates a payment plan for a patient while interacting with that patient and completes the process without the system introducing delays. (M,M)
Maintainability	Routine changes	A hospital increases the fee for a particular service. The configuration team makes the change in 1 working day; no source code needs to change. (H,L)
	Upgrades to commercial components	A maintainer encounters search- and response-time deficiencies, fixes the bug, and distributes the bug fix with no more than 3 person-days of effort. (H,M)
		A reporting requirement requires a change to the report-generating metadata. Change is made in 4 person-hours of effort. (M,L)
		The database vendor releases a new version that must be

Key: Utility
H=high
M=medium
L=low

Utility Tree: Next Steps

- A QA or QA refinement without any ASR is not necessarily an error or omission
 - Attention should be paid to searching for unrecorded ASRs in that area.
- ASRs that rate a (H,H) rating are the ones that deserve the most attention
 - A very large number of these might be a cause for concern: Is the system is achievable?
- Stakeholders can review the utility tree to make sure their concerns are addressed.

Tying the Methods Together

Requirement Documents



Stakeholders Interview

For important stakeholders who have been overlooked



Quality Attribute Workshop

Capture inputs from Stakeholders



PALM (Pedigree Attribute eLicitation Method)

Capture Business goals behind the system



Quality Attribute Utility Tree

Repository of scenarios

Summary

- Architectures are driven by architecturally significant requirements: requirements that will have profound effects on the architecture.
 - Architecturally significant requirements may be captured from requirements documents, by interviewing stakeholders, or by conducting a Quality Attribute Workshop.
- Be mindful of the business goals of the organization.
 - Business goals can be expressed in a common, structured form and represented as scenarios.
 - Business goals may be elicited and documented using a structured facilitation method called PALM.
- A useful representation of quality attribute requirements is in a utility tree.
 - The utility tree helps to capture these requirements in a structured form.
 - Scenarios are prioritized.
 - This prioritized set defines your “marching orders” as an architect.

Thank you.....

Credits

- **Chapter Reference from Text T1:** 16, 17, 18
- Slides have been adapted from Authors Slides
Software Architecture in Practice – Third Ed.
 - Len Bass
 - Paul Clements
 - Rick Kazman

© Len Bass, Paul Clements, Rick Kazman, distributed under Creative Commons Attribution License



Reference Chapter 17
Software Architecture in Practice
Third Edition
Len Bass
Paul Clements
Rick Kazman

Software Architecture

Designing & Documenting the Architecture #1

Designing an Architecture



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal
Module RL6.2

Designing & Documenting the Architecture #1



- Define Architecturally Significant Requirements (ASR)
- Gathering (ASR) from Requirement document, Business goals & Stakeholders
- Capturing ASR in an Utility Tree for further refinement
- Architecture Design strategy and Attribute –Driven Design Method



Architecture Design strategy and Attribute –Driven Design Method

Chapter Outline

Design Strategy

The Attribute-Driven Design Method

The Steps of ADD

Summary



Design Strategy

Design Strategy-



Ideas key to architectural design methods

Idea 1

- Decomposition

Idea 2

- Designing to Architecturally Significant Requirements

Idea 3

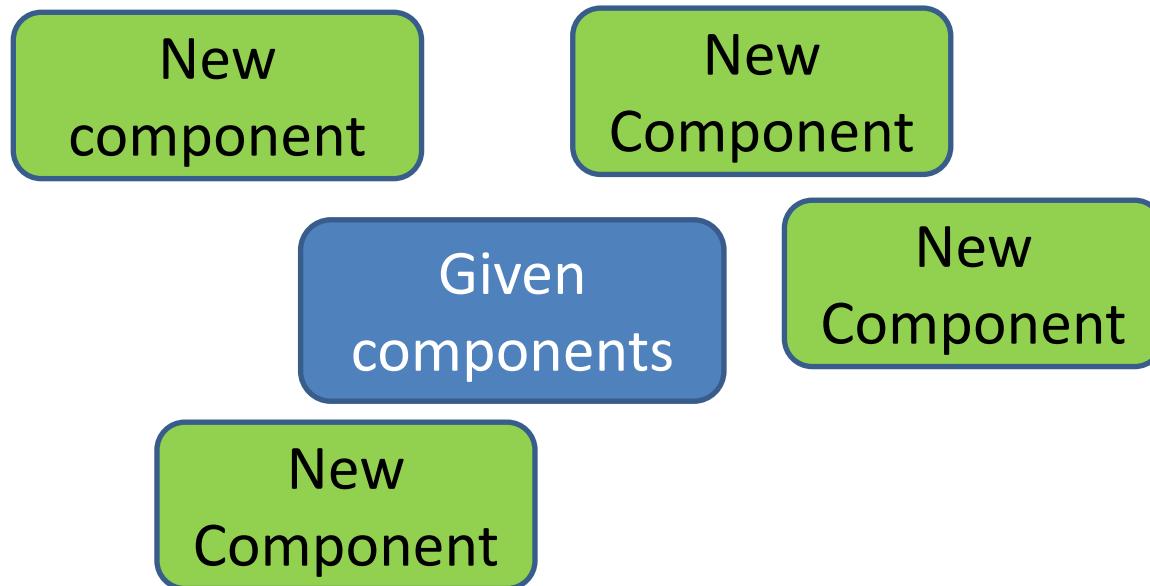
- Generate and Test

Idea 1: Decomposition

- Architecture determines quality attributes
- Important quality attributes are characteristics of the *whole* system.
- Design therefore begins with the whole system
 - The whole system is decomposed into parts
 - Each part may inherit all or part of the quality attribute requirements from the whole

Design Doesn't Mean Green Field

- If you are given components to be used in the final design, then the decomposition must accommodate those components.





Idea 2: Designing to Architecturally Significant Requirements

- Remember architecturally significant requirements (ASRs)?
- These are the requirements that you must satisfy with the design
 - There are a small number of these
 - They are the most important (by definition)

How Many ASRs Simultaneously?

- If you are inexperienced in design then design for the ASRs one at a time beginning with the most important.
- As you gain experience, you will be able to design for multiple ASRs simultaneously.

What About Other Quality Requirements?

If your design does not satisfy a particular non ASR quality requirement then either

- **IMPROVE THE DESIGN**
 - **Adjust your design** so that the ASRs are still satisfied and so is this additional requirement or
- **WEAKEN THE REQUIREMENT**
 - **Weaken the additional requirement** so that it can be satisfied either by the current design or by a modification of the current design or
- **CHANGE PRIORITIES**
 - **Change the priorities** so that the one not satisfied becomes one of the ASRs or
- **DECLARE NON-SATISFIABLE**
 - **Declare the additional requirement non-satisfiable** in conjunction with the ASRs.

Idea 3: Generate and Test

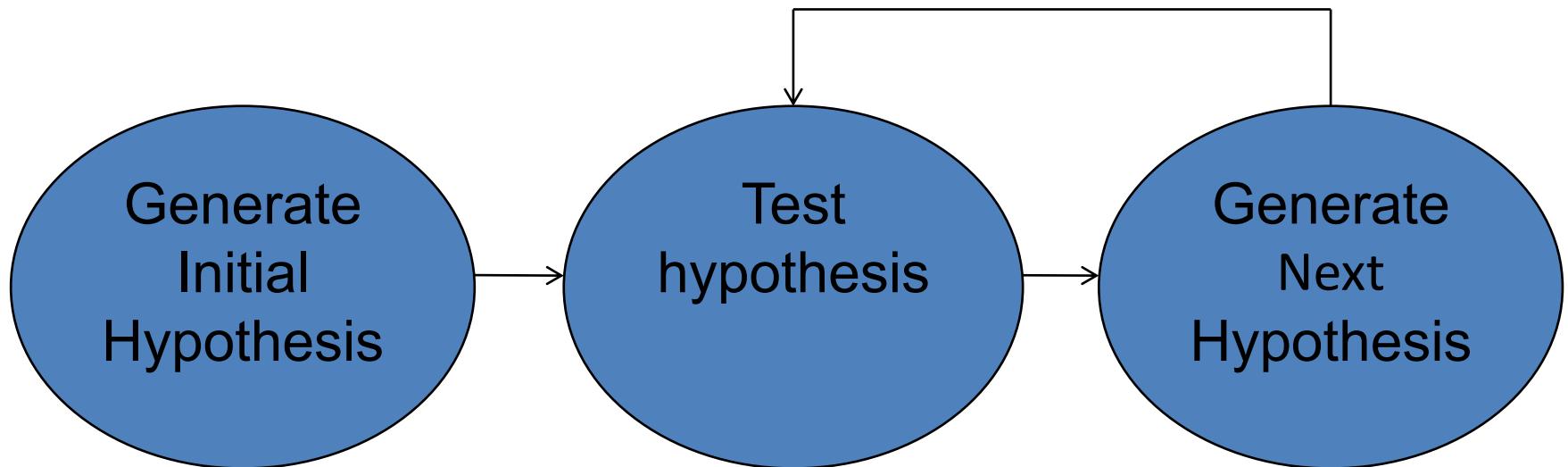
View the current design as a hypothesis.

- Assume requirement will be satisfied

Ask whether the current design satisfies the requirements

- Test if requirement is satisfied.

If not, then generate a new hypothesis

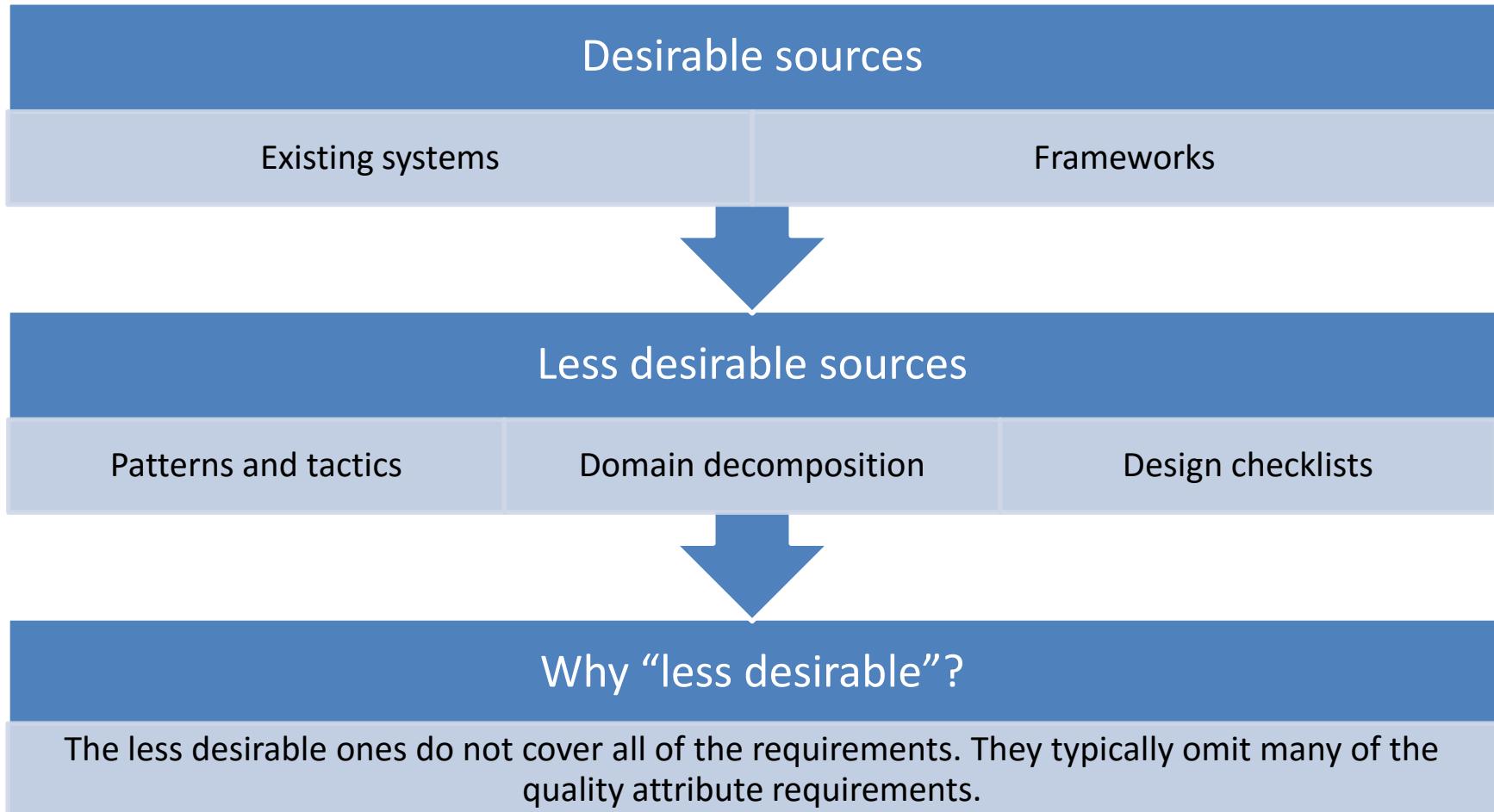


Raises the Following Questions

- Where does initial hypothesis come from?
- How do I test a hypothesis?
- When am I done?
- How do I generate the next hypothesis?

- You already know most of the answers; it is just a matter of organizing your knowledge.

Initial Hypothesis come from “collateral” that are available to the project



How Do I Test a Hypothesis?

Use the analysis techniques already covered

Design checklists from quality attribute discussion.

- Example- coordination model to support capturing activity to support testabilitycollect

Architecturally significant requirements

- Does the hypothesis provide a solution for the ASR.

What is the output of the tests?



List of requirements

either responsibilities

not met by current design

Or quality

not met by current design.

How Do I Generate the Next Hypothesis?



Add missing responsibilities.

Be mindful of the side effects of a tactic.



Use tactics to adjust quality attribute behavior of hypothesis.

The choice of tactics will depend on which quality attribute requirements are not met.

When Am I Done?

All ASRs are satisfied
and/or...

You run out of budget for
design activity

- In this case, use the best hypothesis so far.
- Begin implementation
- Continue with the design effort although it will now be constrained by implementation choices.



The Attribute-Driven Design Method

The Attribute-Driven Design Method

Packaging of many of the techniques already discussed.

An iterative method. At each iteration you

- Choose a part of the system to design.
- Marshal all the architecturally significant requirements for that part.
- Generate and test a design for that part.

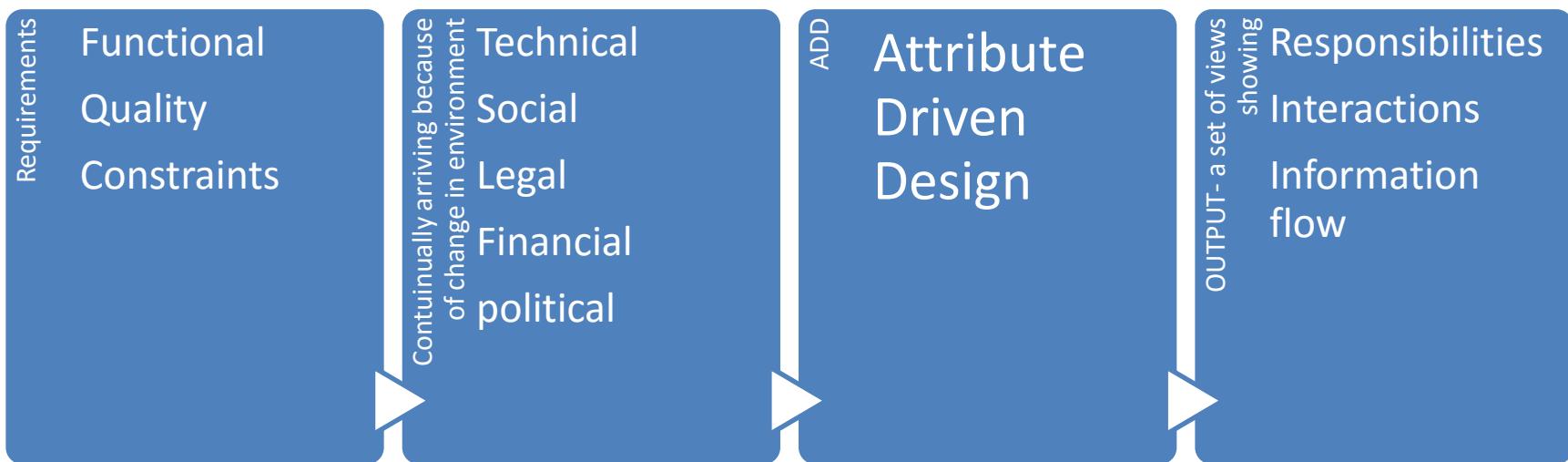
ADD does not result in a complete design but the main design approach

- Set of containers with responsibilities
- Interactions and information flow among containers

Does not produce an API or signature for containers.

- Gives a “workable” architecture early and quickly

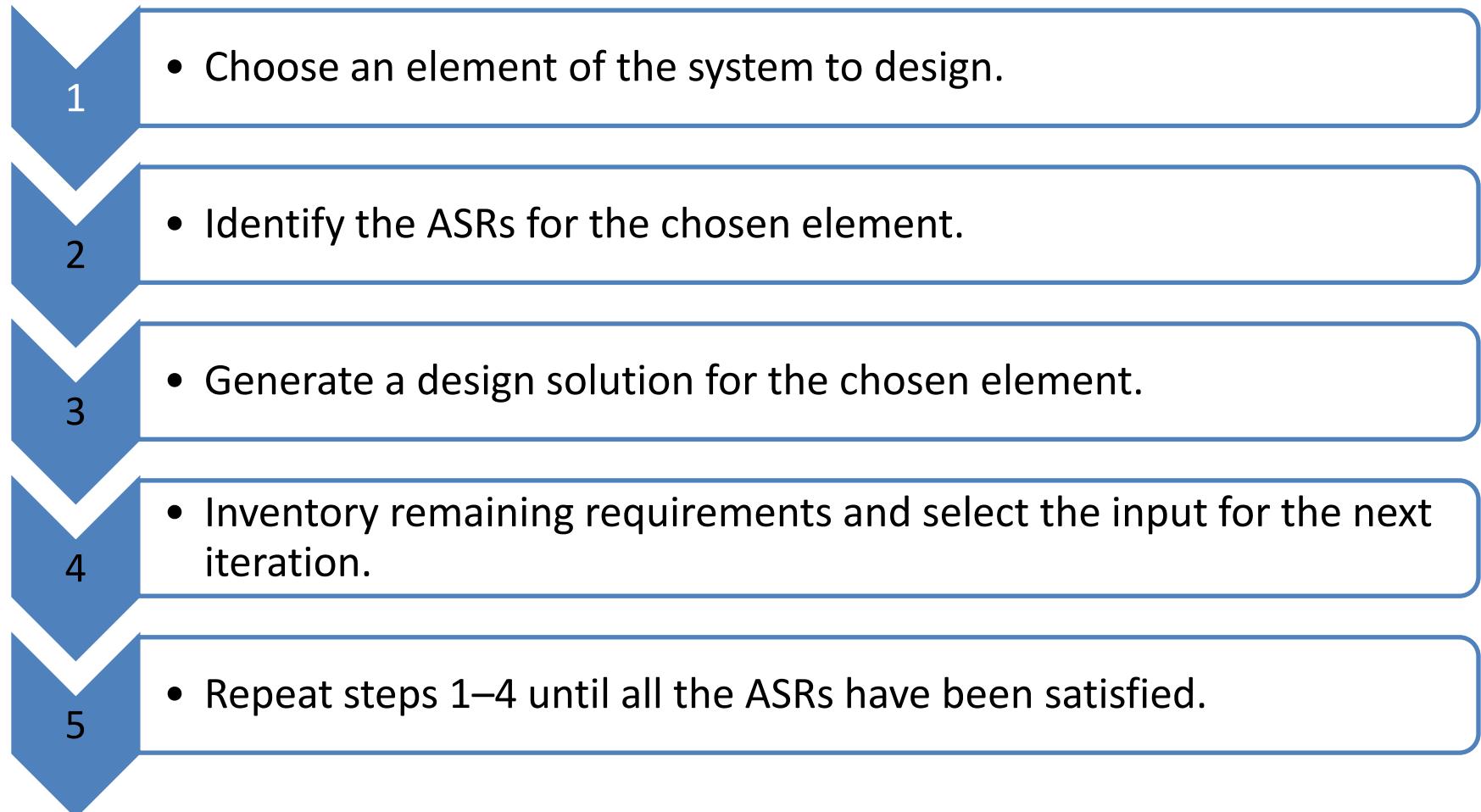
ADD Inputs and Outputs





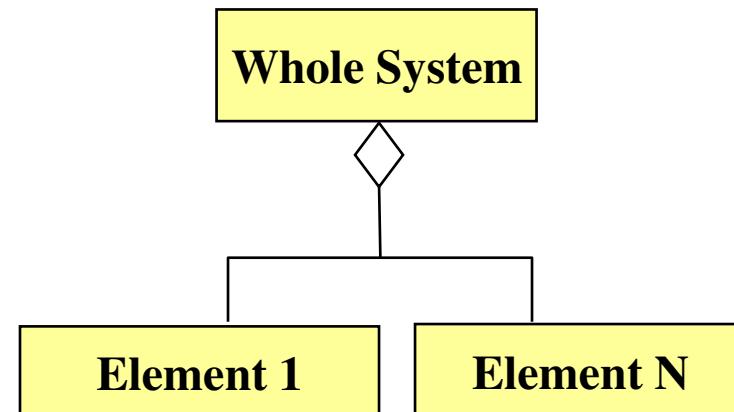
The Steps of ADD

The Steps of ADD



- Choose an Element of the System to Design

- For green field designs, the element chosen is usually the whole system.
- For legacy designs, the element is the portion to be added.
- After the first iteration:



- *Initial iteration will be broad with less depth.*
- *Gradually get fine-grained.*

Which Element Comes Next?

Two basic refinement strategies:

Breadth first

Depth first

Which one to choose?

It depends ☺

If using new technology
=>

depth first: explore the implications of using that technology.

If a team needs work
=>

depth first: generate requirements for that team.

Otherwise
=>

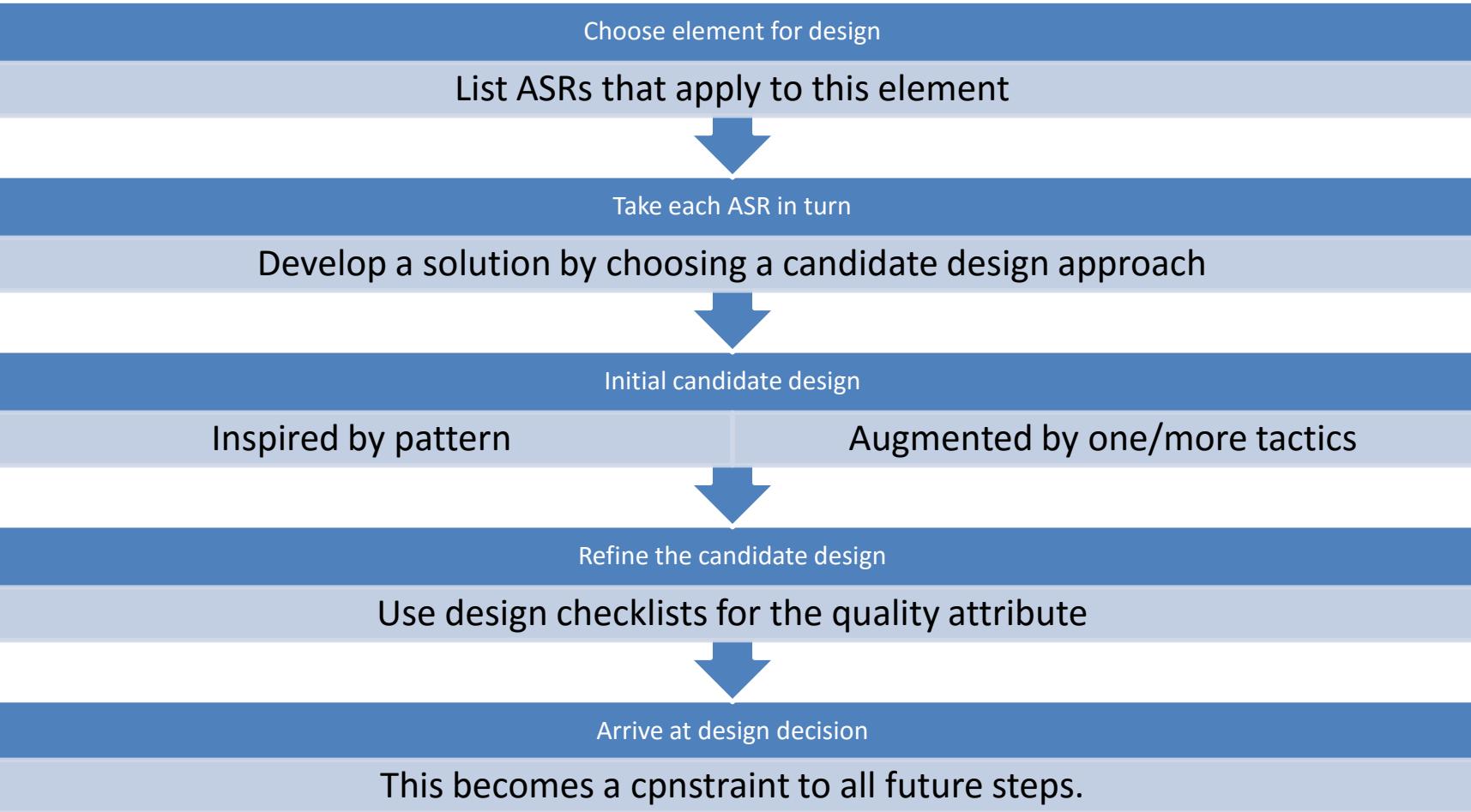
breadth first.

- Identify the ASRs for the Chosen Element

- If the chosen element is the whole system, then use a utility tree (as described earlier).
- If the chosen element is further down the decomposition tree, then generate a utility tree from the requirements for that element.

Step 3

- Generate a Design Solution for the Chosen Element



• Select the Input for the Next Iteration

Ensure that requirement has been satisfied,

- if not:
- BACKTRACK.

ASR not yet satisfied should relate to

- Quality Attribute Requirement/
- Functional responsibility /
- constraint of the parent element

then add responsibilities to satisfy the requirement.

- Add them to container with similar requirements
- If no such container may need to create new one or add to container with dissimilar responsibilities (coherence)
- If container has too many requirements for a team, split it into two portions. Try to achieve loose coupling when splitting.

For each Quality Attribute Requirements, responsibility and constraint.

If the quality attribute requirement has been satisfied,

- it does not need to be further considered.

If the quality attribute requirement has not been satisfied then either

- Delegate it to one of the child elements
- Split it among the child elements

If the quality attribute cannot be satisfied,

- see if it can be weakened.
- If it cannot be satisfied or weakened then it cannot be met.

Constraints

Constraints are treated as quality attribute requirements have been treated.

Satisfied

Delegated

Split

Unsatisfiable

- Repeat Steps 1–4 Until All ASRs are Satisfied

At end of step 3, each child element will have associated with it a set of:

functional requirements
(responsibilities),

quality attribute requirements,
and

constraints.



This sets up the child element for the next iteration of the method.



ADD PROCESS CAN BE TERMINATED IF

All
requirements
satisfied

High degree of trust
between architect and
implementation team.

Contractual
arrangement satisfied.

Project's design budget
exhausted.



Summary

Summary

Designing the architecture is a matter of

Determining
the ASRs

Performing
generate and
test one an
element to
decompose it to
satisfy the ASRs

Iterating until
requirements
are satisfied.



Thank you.....

Credits

- **Chapter Reference from Text T1:** 16, 17, 18
- Slides have been adapted from Authors Slides
Software Architecture in Practice – Third Ed.
 - Len Bass
 - Paul Clements
 - Rick Kazman



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 8.2): Software Architecture

Introduction to Agile Methodology

Instructor: Prof. Santonu Sarkar

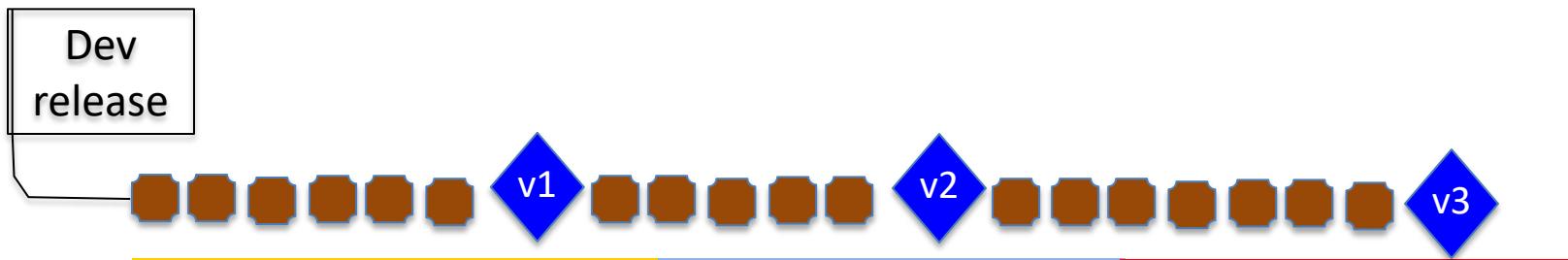
What is Agile Methodology

- Collaborative
 - Forms a pair for any development task to avoid error
 - Involves stakeholders from the beginning
- Interactive and feedback oriented
 - Teams interact frequently
 - Quick, and repeated integration of the product
 - Constant feedback from the stakeholder (customer)
- Iterative
 - Requirement, design, coding, testing goes through many iterations each having short duration
 - Refactoring is a part of the development process
- Test driven
 - Before building the component, define the test cases
 - Continuously test

– Scott Amberg, Kent Beck

A Brief Overview

- There are 7 disciplines performed in an iterative manner
- At each iteration the software (or a part of the software) is built, tested
- Software architecture is more “agile” and it is never frozen
- UML based modeling is performed



Discipline Overview

- Model
 - Business Model
 - Analysis and Design (Architecture)
 - Implementation
 - Test
 - Deployment
 - Config Management
 - Project Management
 - Environment
-



Steps of Architecture Modeling in Agile

- Feature driven
 - Prioritize. Elaborate critical features more
- Model the architecture (UML)
- Suggested viewpoints for Agile
 - Usage scenarios
 - User interface and system interface
 - Network, deployment, hardware
 - Data storage, and transmission
 - Code distribution
- Suggested quality concerns
 - Reuse
 - Reliability, availability, serviceability, performance
 - Security
 - Internationalization, regulation, maintenance

Class Responsibilities and Collaborators (CRC) Card

What

- It is a physical (electronic) card
- One card for one class
 - Indicates the responsibilities of a class
- Collaboration
 - Sometimes a class can fulfill all its assigned responsibilities on its own
 - But sometimes, it needs to collaborate with other classes in order to fulfill its own responsibilities

Why?

- A good technique to identify a class and its responsibility during functional architecture design
- Highly collaborative and interactive process for a team of designers
- The team can do it fast



CRC Card

Class Name	Collaborators
Responsibilities assigned to this Class	If this class can not fulfill any of its assigned task on its own then which other classes it has to collaborate

CRC Card Example 1

```

class Box
{
    private double length;
    private double width;
    private double height;
    Box(double l, double w, double h) { length = l; width = w; height = h; }
    public double getLength() { return length; }
    public double getWidth() { return width; }
    public double getHeight() { return height; }
    public double area() { return 2*(length*width + width * height + height * length); }
    public double volume() { return length * width * height; }
} // End of class BOX

```

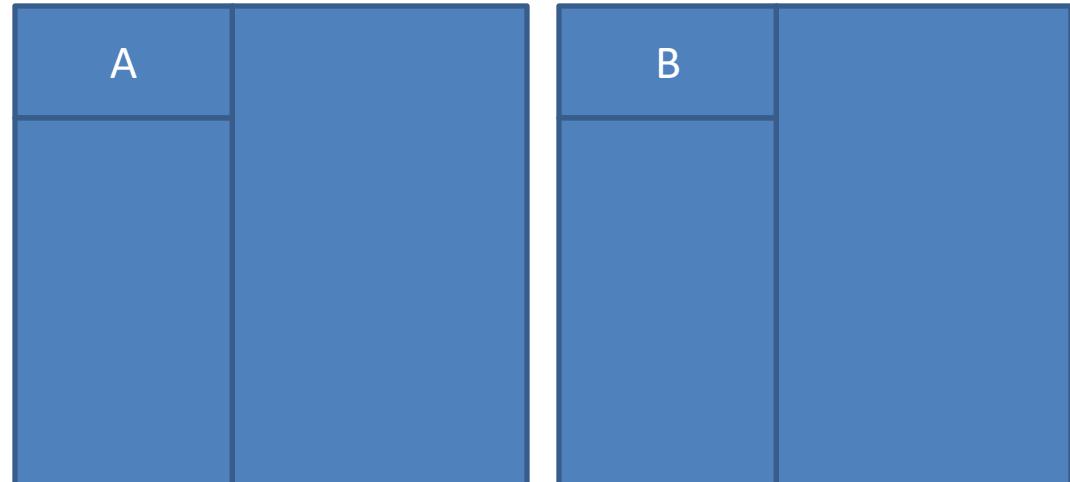
Write CRC Cards for Box Class

Box	Collaborators
Responsibilities	<<None>>
<ol style="list-style-type: none"> 1. Getting length 2. Getting width 3. Getting height 4. Computing area and volume 	

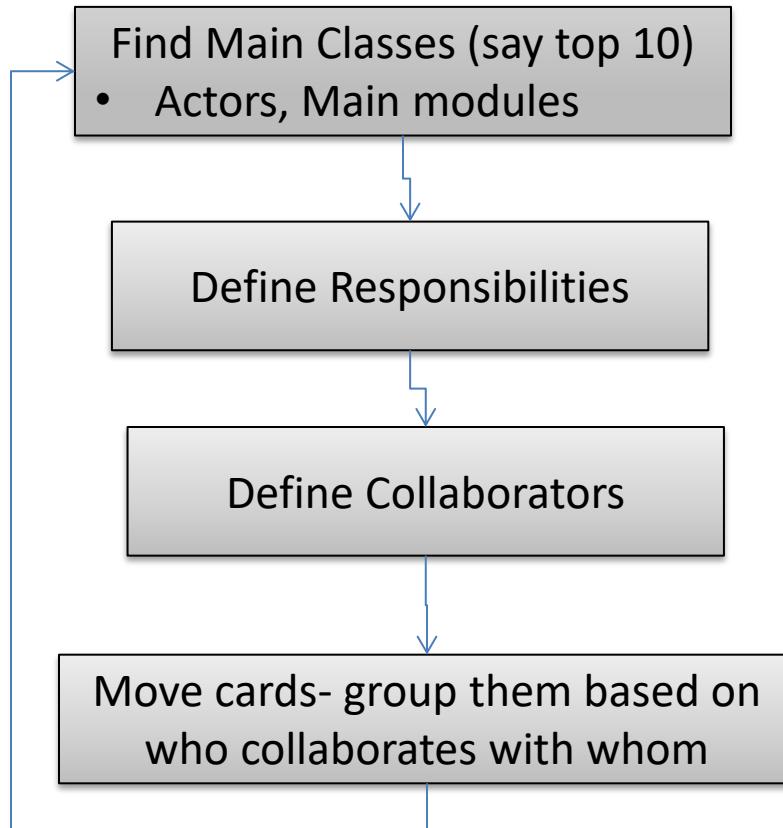
CRC Card Example 2

```
class B
{
    public void doB()
    {
        System.out.println("Hello");
    }
}
class A
{
    public void doS()
    {
        B b1 = new B();
        b1.doB();
    }
} //End of class Test
```

Write CRC Cards for Classes A & B



How do you create CRC Model?



Thank You



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 7.1): Software Architecture

Introduction to OODesign

Instructor: Prof. Santonu Sarkar

Online shopping application- Pet Store

by Sun Microsystems (Oracle)

- Storefront has the main user interface in a Web front-end. Customers use the Storefront to place orders for pets
 - Register User
 - Login user
 - Browse catalog of products
 - Place order to OPC (asynchronous messaging)
 - Order Processing Center (OPC) receives orders from the Storefront.
 - Administrators
 - Examine pending orders
 - Approve or deny a pending order
 - Supplier
 - View and edit the inventory
 - Fulfills orders from the OPC from inventory and invoices the OPC.
-

System quality

- Availability
 - Show orders in multiple languages
 - Help in browsing products
 - Easy checkout
 - Assurance to customer, supplier and bank
 - Guaranteed delivery
 - Shopping cart is only modified by the customer
 - Order never fails
 - System is always ready to sell
 - Order is always processed in 2sec.
 - System always optimally use the hardware infrastructure and performs load-balancing
 - Uses standard protocol to communicate with Suppliers
 - Uses secure electronic transaction protocol (SET) for credit card processing
- Testability
 - Quite easy to add new supplier
 - Little change necessary to add a third party vendor
 - Quite easy to sell books in addition to Pets
 - Logging is extensive to trace the root cause of the fault
- Usability
 - Quite easy to add new supplier
 - Little change necessary to add a third party vendor
 - Quite easy to sell books in addition to Pets
 - Logging is extensive to trace the root cause of the fault
- Interoperability
 - Quite easy to add new supplier
 - Little change necessary to add a third party vendor
 - Quite easy to sell books in addition to Pets
 - Logging is extensive to trace the root cause of the fault

Example

Programming for this Application

- Everything is an object
 - Pet, Customer, Supplier, Credit card, Customer's address, order processing center
- Pets can be sold, credit card can be charged, clients can be authenticated, order can be fulfilled!
 - Have behavior, properties
- Interact with each other
 - Storefront places order to OPC

What's cool?

- Very close to the problem domain... domain experts can pitch in easily.
- Could group features and related operations together
- It's possible to add new types of Pets (e.g., add Birds in addition to Cat, Dog and Fish)

What is a class

- Class represents an abstract, user-defined type
 - Structure – definition of properties/attributes
 - Commonly known as member variables
 - Behavior – operation specification
 - Set of methods or member functions
 - An object is an instance of a class. It can be instantiated (or created) w/o a class
-

Object State

-
- Properties/Attribute Values at a particular moment represents the state of an object
 - Object may or may not change state in response to an outside stimuli.
 - Objects whose state can not be altered after creation are known as immutable objects and class as immutable class [**String class in Java**]
 - Objects whose state can be altered after creation are known as mutable objects and class as mutable class [**StringBuffer class in Java**]

States of Three Different INSTANCES of “Dog” class

Labrador

Age: 1 yr

Price : 3500

Sex:Male

Golden Retriever

Age: 6months

Price : 2000

Sex:Female

Pug

Age: 1.5yr

Price : 1500

Sex: Female



Object-Oriented Programming

- A program is a bunch of objects telling each other what to do, by sending messages
 - In Java, C++, C# one object (say o1) invokes a method of another object (o2), which performs operations on o2
 - You can create any type of objects you want
- OO different from procedural?
 - No difference at all: Every reasonable language is ultimate the same!!
 - Very different if you want to build a large system
 - Increases understandability
 - Less chance of committing errors
 - Makes modifications faster
 - Compilers can perform stronger error detections

Type System

- Classes are user-defined data-types
- Primitive types
 - int, float, char, boolean in Java (bool in C#), double, short, long, string in C#
- Unified type
 - C# keyword **object** – mother of all types (root)
 - Everything including primitive types are objects
 - Java JDK gives **java.lang.Object** – not a part of the language
 - Primitive types are not objects

References and Values

Value

- Primitive types are accessed by value
- C++ allow a variable to have object as its value
- C# uses **struct** to define types whose variables are values
- No explicit object creation/deletion required
 - Faster, space decided during compilation

Reference

- Java allows a variable to have reference to an object only
- C++ uses pointers for references
- Needs explicit object creation
 - Slower, space allocated during runtime from heap
- Java performs escape analysis for faster allocation

Modules

- You need an organization when you deal with large body of software – 20MLOC, 30000 classes or files!
- Notion of module introduced in 1970
 - Group similar functionality together
 - Hide implementation and expose interface
 - Earlier languages like Modula, Ada introduced this notion
 - In OO language “class” was synonymous to a module
- But they all faced the problem of managing 20M lines of C or C++ code
- ANSI C++, Java, Haskell, C#, Perl, Python, PHP all support modules
 - Namespace (C++, C#)
 - Package (Java, Perl)

Module- aka namespace, package

- A set of classes grouped into a module
 - A module is decomposed into sub-modules
 - Containment hierarchy of modules – forms a tree
 - A fully qualified, unique name= module+name of the class
 - namespace
 - package
 - Import- A class can selectively use one or more classes in a module or import the entire module
-

Inheritance

- Parent (called Base class) and children classes (Derived classes)
 - A Derived class inherits the methods and member variables of the Base
-- also called ISA relationship
 - A child can have multiple parents – Multiple inheritance
 - Hierarchy of inheritance (DAG)
- The Derived class can
 - Use the inherited variables and methods – reuse
 - Add new methods – extends the functionality
 - Modify derived method- called **overriding** the base class member function

Introducing Interface

What is it?

- A published declaration of a set of services
 - An interface is a collection of constants and method declarations
 - No implementation, a separate class needs to implement an interface
- A class can implement more than one interfaces

Why?

- Provide capability for unrelated classes to implement a set of common methods
- Standardize interaction
- Extension- let's the designer to defer the design
- User does not know who implemented it
 - It is easy to change implementation without impacting the user

Interface Definition

- An interface in C++, C# is a class that has at least one pure virtual method
 - A pure virtual method only has specification, but no body
 - This is called abstract base class
 - One can have an abstract class where some methods are concrete (with implementation) and some as pure virtual which could be clumsy
 - Java uses keyword “interface” and is more clean
 - Interface only provides method declarations
 - Java also allows to define an abstract base class just like C++
-

Creating Objects

- With built-in types like **int** or **char**, we just say
int i; char c; --- and we get them
- When we define a class A-- user-defined type
 - We need to explicitly
 1. tell that we want a new object of type A (operator **new**)
 2. Initialize the object (you need to decide) after creation
 - constructor method
 - Special method that compiler understands. The constructor method name must be same as the class name
 - C++ allows constructor method for struct also
 - Constructor methods can be overloaded

Destroying Objects

- If an object goes “out of scope,” it can no longer be used (its name is no longer known) it is necessary to free the memory occupied by the object
 - Otherwise there will be a “memory leak”
1. Just before freeing the memory
 - It is necessary to perform clean-up tasks (you need to decide) just before getting deleted
 - » Destructor method describes these tasks
 - » Special name for destructor method ~<ClassName> in C++
 - » `finalize()` method in Java
 - » does not have any return value
 2. Then free the memory
 - In C++, we need to explicitly delete this object (`delete` operator)
 - Java uses references and “garbage collection” automatically.

Thank You

Next class - UML



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 7.2): Software Architecture

Introduction to UML

Instructor: Prof. Santonu Sarkar

Unified Modeling Language (Introduction)

- **Modeling Language for specifying, Constructing, Visualizing and documenting software system and its components**
- Model -> Abstract Representation of the system [Simplified Representation of Reality]
- UML supports two types of models:
 - Static
 - Dynamic

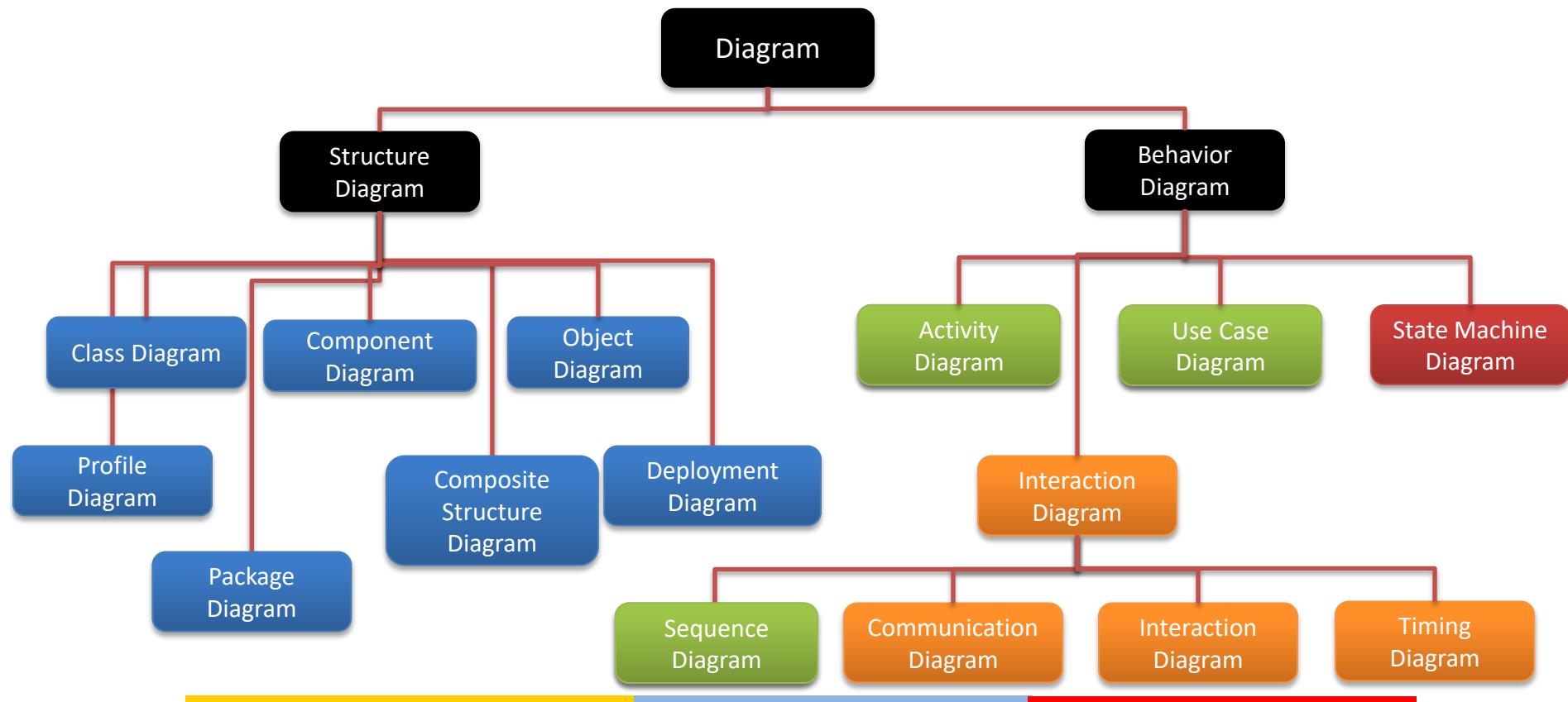


UML

- **Unified Modeling Language is a standardized general purpose modeling language in the field of object oriented software engineering**
 - The standard is managed, and was created by, the **Object Management Group**.
 - UML includes a set of graphic notation techniques to create visual models of object-oriented software-intensive systems
-

UML Diagrams overview

- Structure diagrams emphasize the things that must be present in the system being modeled- extensively used for documenting software architecture
- Behavior diagrams illustrate the behavior of a system, they are used extensively to describe the functionality of software systems.



Structural Diagrams

- **Class diagram:** system's classes, their attributes, and the relationships
 - Component diagram: A system, comprising of components and their dependencies
 - Composite structure diagram: decomposition of a class into more finer elements and their interactions
 - **Deployment diagram:** describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.
 - Object diagram: shows a complete or partial view of the structure of an example modeled system at a specific time.
 - Package diagram: describes how a system is split up into logical groupings by showing the dependencies among these groupings.
 - Profile diagram: operates at the metamodel level
-

Behavioral Diagrams

- **Activity diagram:** describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.
 - **State machine diagram:** describes the states and state transitions of part of the system.
 - **Use case diagram:** describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases
-

Behavioral Model- Interactions

- Communication diagram: shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system.
- Interaction overview diagram: provides an overview in which the nodes represent communication diagrams.
- **Sequence diagram:** Interaction among objects through a sequence of messages. Also indicates the lifespans of objects relative to those messages
 - Timing diagrams: a specific type of sequence diagram where the focus is on timing constraints.

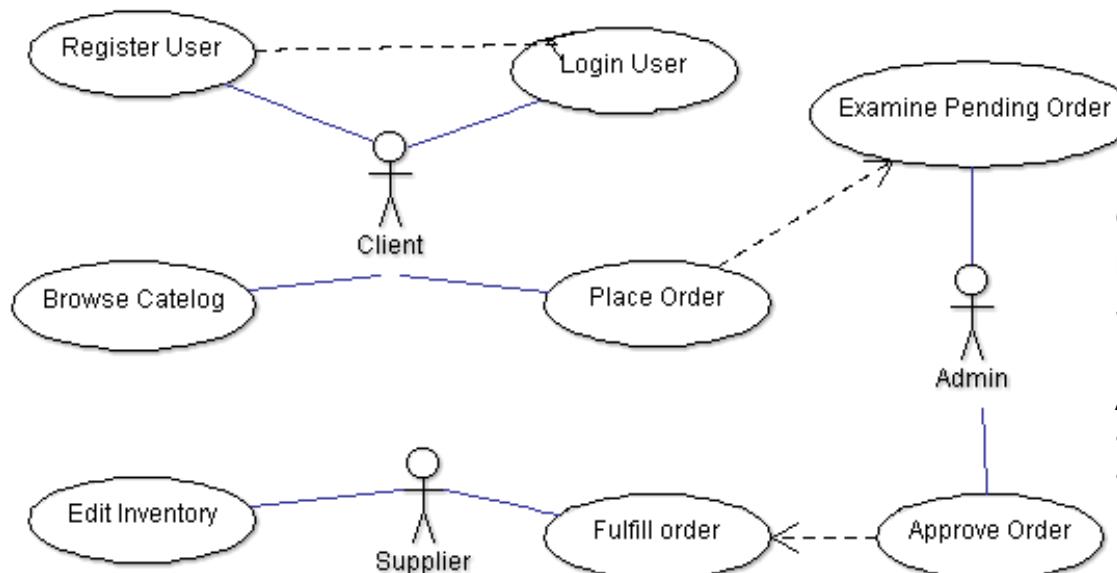
Petstore Shopping System



Use Case

Storefront has the main user interface in a Web front-end. Customers use the Storefront to place orders for pets

- Register User
- Login user
- Browse catalog of products
- Place order to OPC (asynchronous messaging)



Order Processing Center (OPC) receives orders from the Storefront.

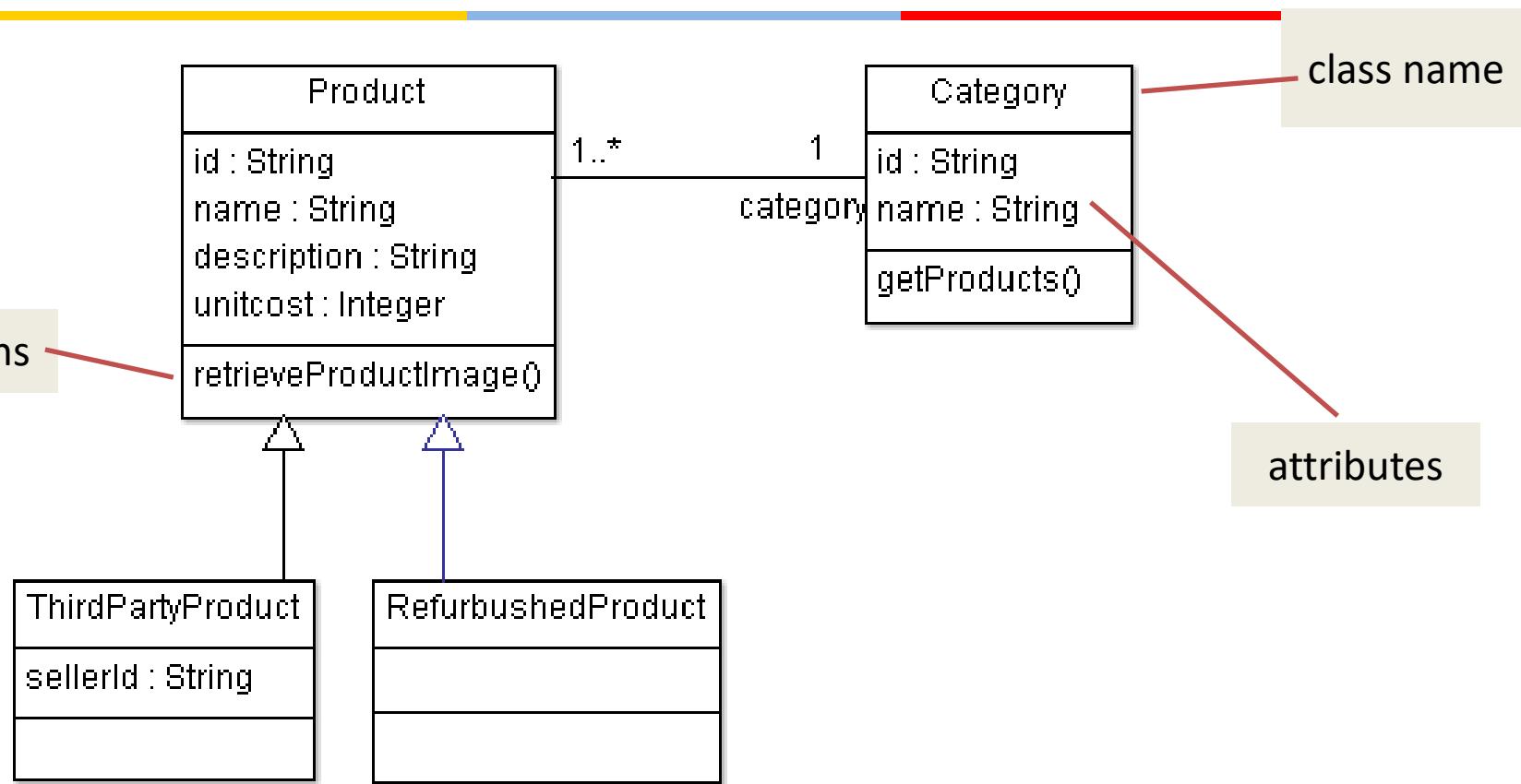
Administrator

- Examine pending orders
- Approve or deny a pending order

Supplier

- View and edit the inventory
- Fulfils orders from the OPC from inventory and invoices the OPC.

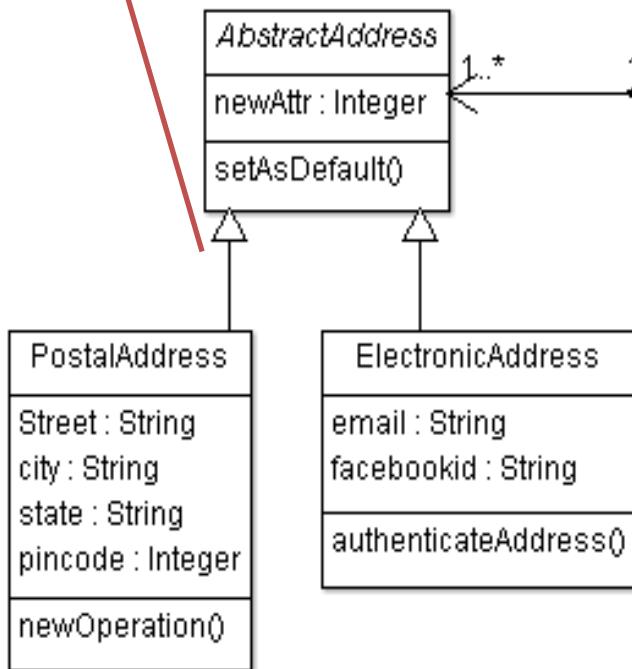
Class Notation



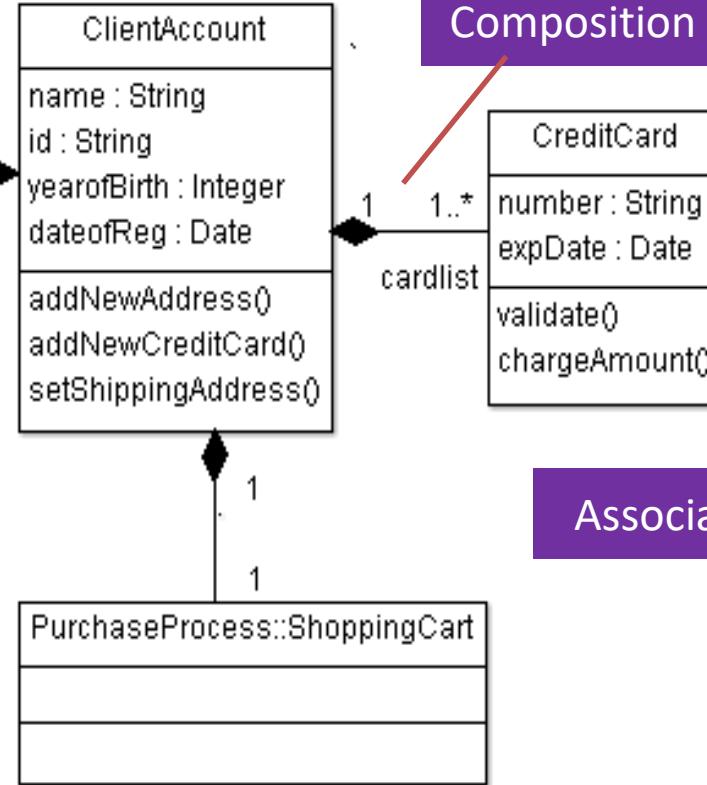
Visibility Modes : Private(-), Protected (#) , Public (+) and Package Private()

Class Relationships

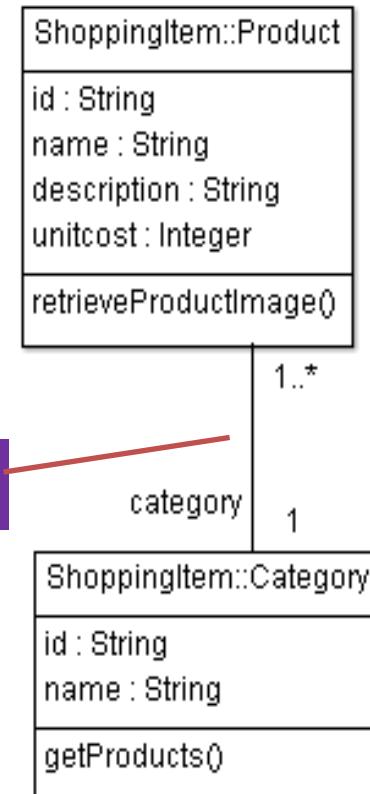
Inheritance



Composition



Association



Generalization Example

- isA, is-a-type of relation ship
- A PostalAddress, or an EmailAddress is an Address
- There can be ThirdPartyProduct or a Refurbished product in the online store

```
public class PostalAddress extends AbstractAddress {  
  
    private String Street;  
    private String city;  
    private String state;  
    private Integer pincode;  
}
```

```
public class ElectronicAddress  
extends AbstractAddress {  
  
    private String email;  
    private String facebookid;  
  
    public void authenticateAddress() {  
    }  
}
```

Different forms of association

1. Strongest (Composition)

- Implies total ownership, if the owner is destroyed, the parts are also destroyed
- Inner classes will certainly be a composition

2. Aggregation

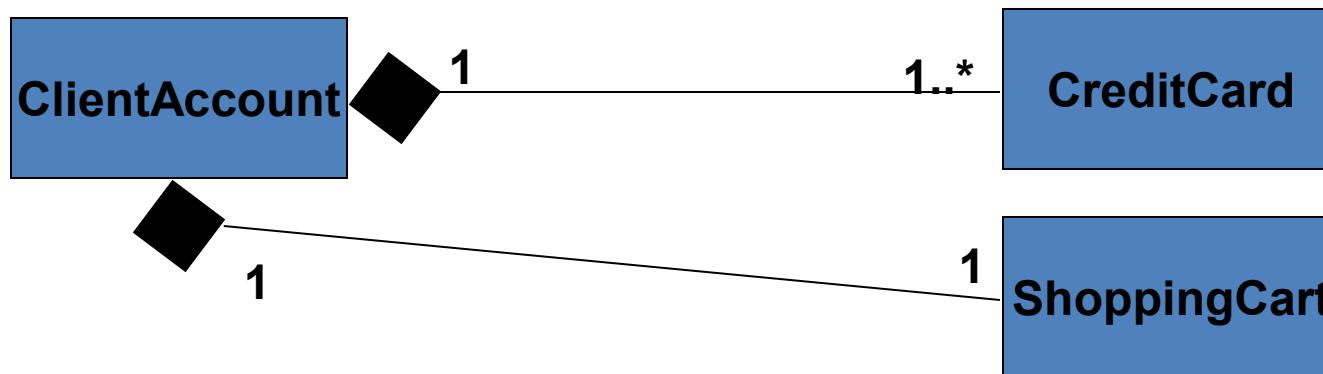
- Implies has-a part ownership
- If the owner is destroyed, the parts still exist

3. Weakest (Association)

- General form of dependency based on the usage of features

Composition

- Total ownership
- A CreditCard exclusively belongs to one Client, and one client can have many credit cards.
- A client exclusively owns her shopping cart.
- The shopping cart and the credit card will no longer exist if the client is removed



Composition Code snippet

```
public class ClientAccount {
    private String name;
    private String id;
    private Integer yearOfBirth;
    private Date dateofReg;
private Vector myAbstractAddress;
private Vector myCreditCard;
private ShoppingCart myShoppingCart;

    public void addNewAddress() { }
    public void addNewCreditCard() { }
    public void setShippingAddress() { }
}
```

```
public class CreditCard {
    private String number;
    private Date expDate;
private ClientAccount
myClientAccount;

    public void validate() { }
    public void chargeAmount() { }
}
```

```
public class ShoppingCart {
private Vector myShoppingCartController;
private ClientAccount myClientAccount;
}
```

Aggregation Relationship

- Relatively weaker composition
- Students will exist even when the Professor stops taking the class
- Ducks will exist without the Pond



Association or Dependency Relation

- A product category in the online shop can have many products
- However, a product belongs to only one category.
- Both of them independently exist.



```
public class Product {  
    private String id;  
    private String name;  
    private String description;  
    private Integer unitcost;  
    private Category category;  
  
    public void retrieveProductImage() {  
    }  
}
```

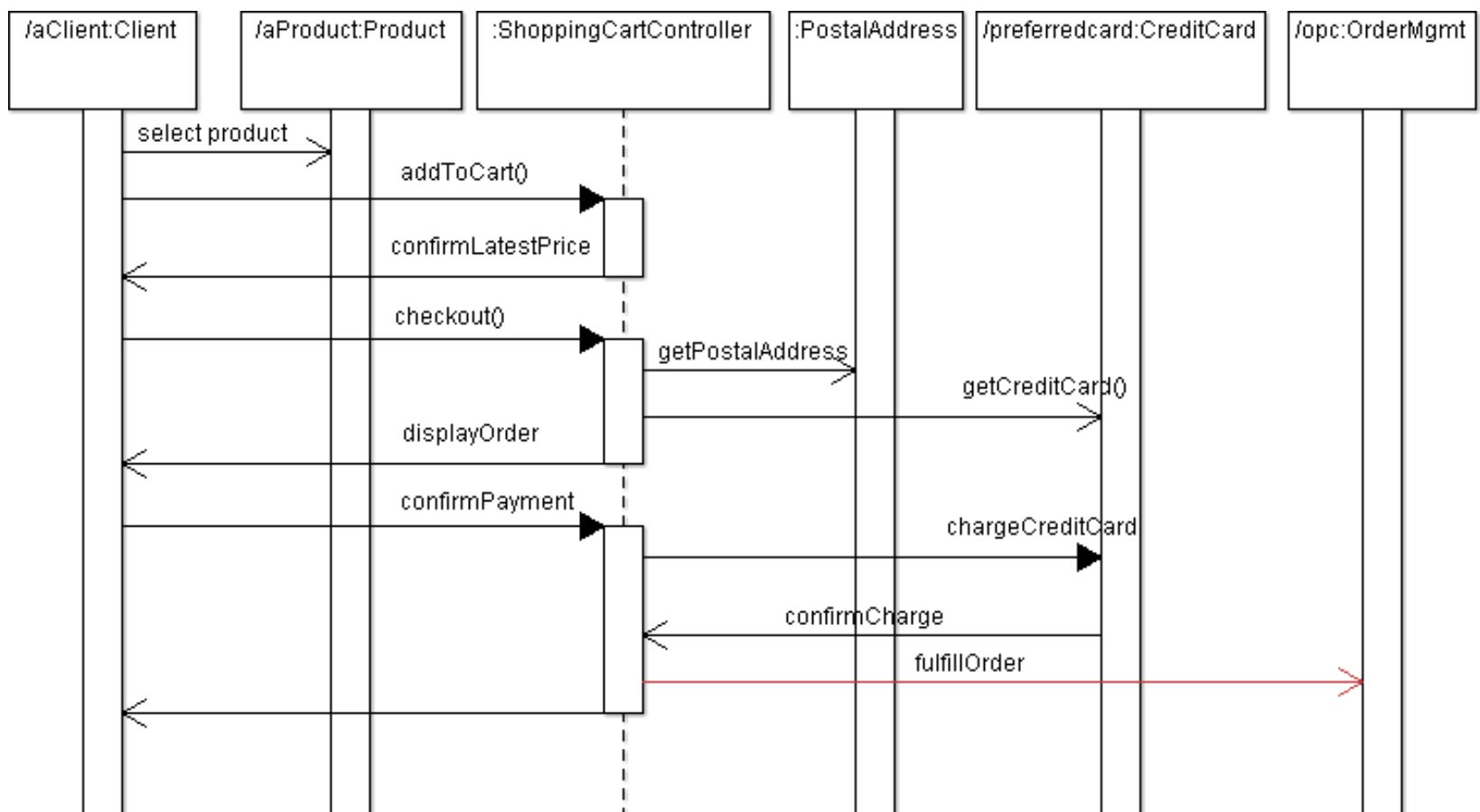
```
public class Category {  
    private String id;  
    private String name;  
    private Vector<Product> myProduct;  
  
    public void getProducts() {  
    }  
}
```

Dependency Example 2

```
class B
{
    public void doB()
    {
        System.out.println("Hello");
    }
}
class A
{
    public void doS()
    {
        B b1 = new B();
        b1.doB();
    }
} //End of class Test
```



Sequence Diagrams



Sequence diagram is drawn to represent (i) objects participating in an interaction and (ii) what messages have exchanged among those objects

Thank You



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 8.1): Software Architecture

Documenting Architecture with UML

Instructor: Prof. Santonu Sarkar

Unified Modeling Language (Introduction)

- **Modeling Language for specifying, Constructing, Visualizing and documenting software system and its components**
- Model -> Abstract Representation of the system [Simplified Representation of Reality]
- UML supports two types of models:
 - Static
 - Dynamic

UML

- **Unified Modeling Language is a standardized general purpose modeling language in the field of object oriented software engineering**
- The standard is managed, and was created by, the **Object Management Group**.
- UML includes a set of graphic notation techniques to create visual models of object-oriented software-intensive systems

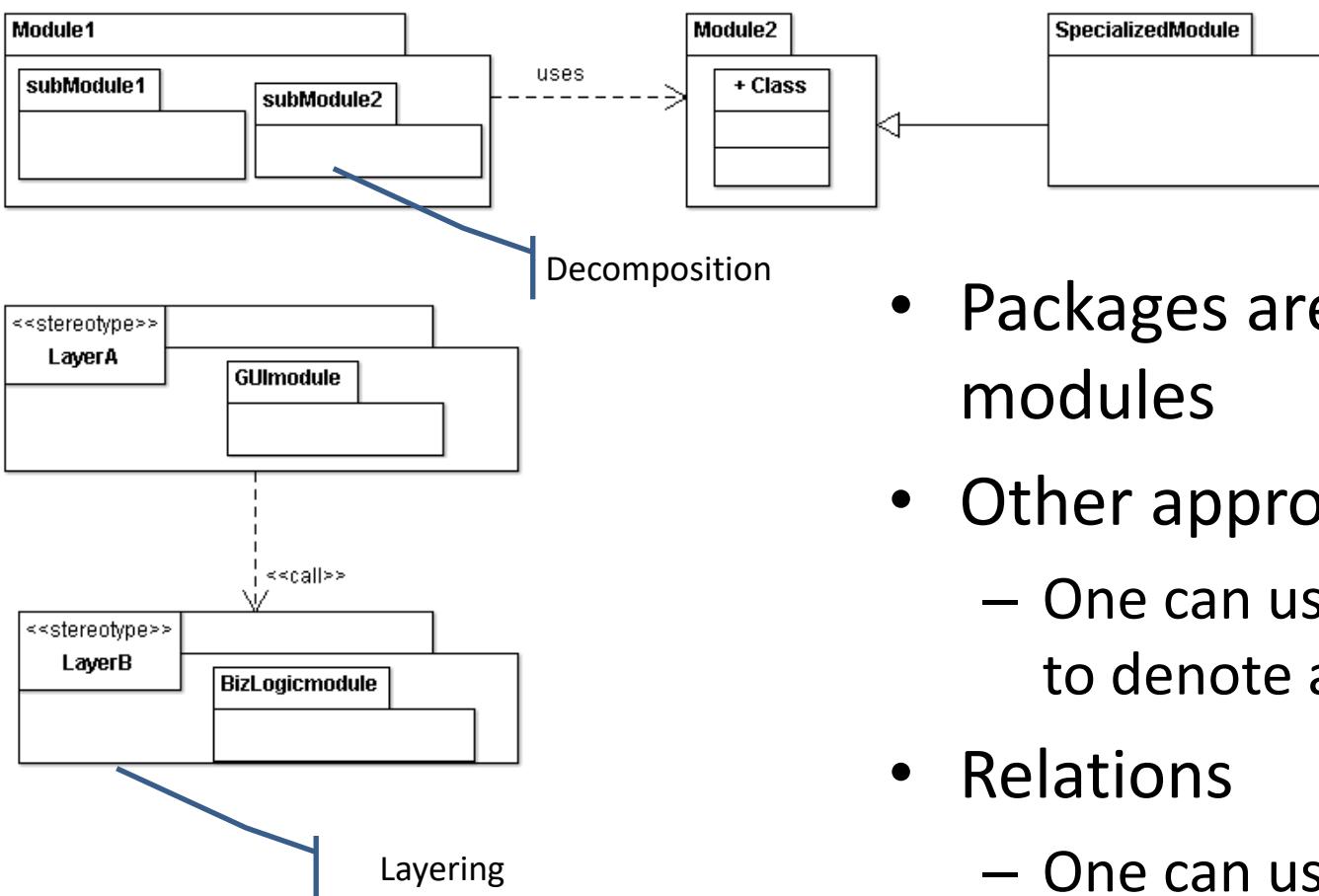
Documenting Architecture

- UML has not been designed specifically for architecture, though practitioners use UML for architecture description
 - It is up to the architect to augment UML for architecture
 - UML provides no direct support for module-structure, component-connector structure or allocation structure

Three Structures- Recap

- Module Structure
 - Code units grouped into modules
 - Decomposition
 - Larger modules decomposed into smaller modules
 - Use
 - One module uses functionality of another module
 - Layered
 - Careful control of uses relation

Illustration- Module Views

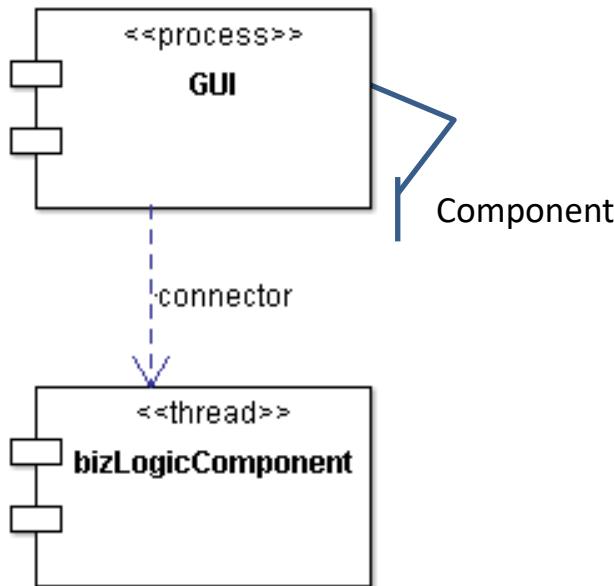


- Packages are used as modules
- Other approaches
 - One can use class, or interface to denote a module
- Relations
 - One can use various other UML relations to denote uses, layering or generalization

Three Structures- Recap

- Component-Connector Structure
 - Processes
 - Components are processes and relations are communications among them
 - Concurrency
 - Relationships between components- control flow dependency, and parallelism
 - Client-Server
 - Components are clients or servers, connectors are protocols
 - Shared data
 - Components have data store, and connectors describe how data is created, stored, retrieved

Illustration- CNC Views



- No standard representation exists
- UML components are used with stereotypes
- Other approaches
 - One can use class, interface, or package to denote a component
- Relations
 - One can use various other UML relations such as association class

Three Structures- Recap

- Allocation Structure

- Deployment

- Units are software (processes from component-connector) and hardware processors
 - Relation means how a software is allocated or migrated to a hardware

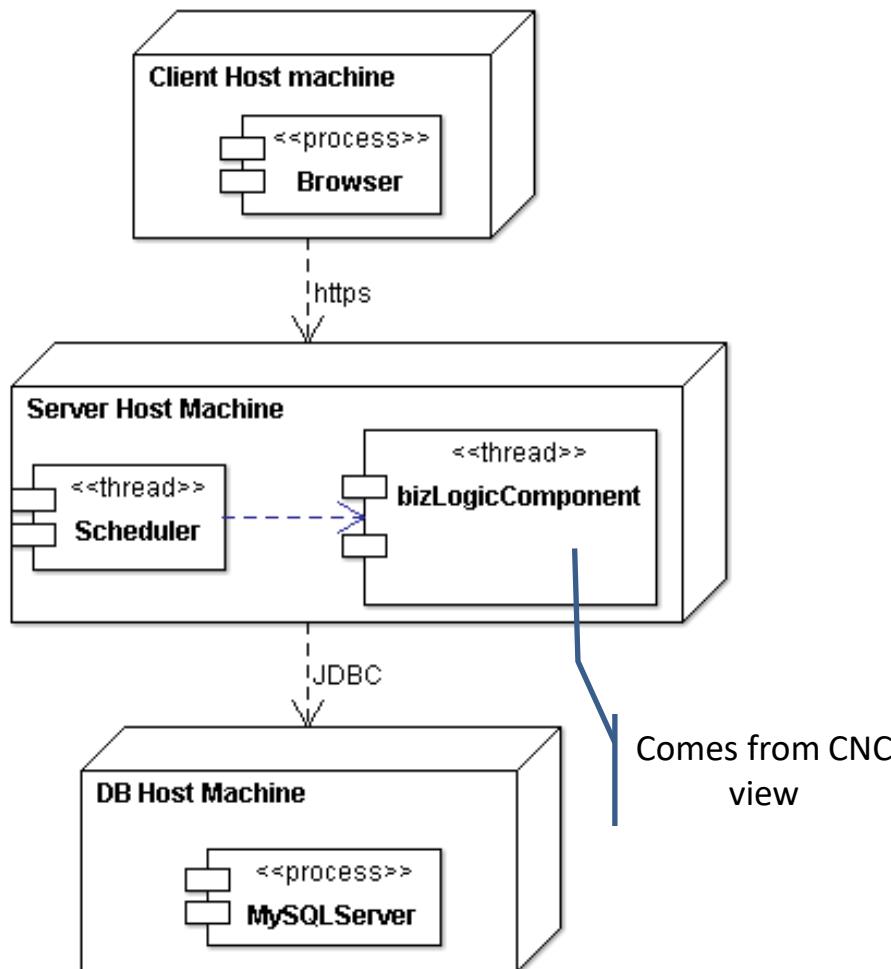
- Implementation

- Units are modules (from module view) and connectors denote how they are mapped to files, folders

- Work assignment

- Assigns responsibility for implementing and integrating the modules to people or team

Illustration-Allocation Views



- UML Deployment diagram is a good option for deployment structure
- No specific recommendation for work assignment and implementation

Thank You



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 8.3): Software Architecture

Introduction to Unified Process

Instructor: Prof. Santonu Sarkar

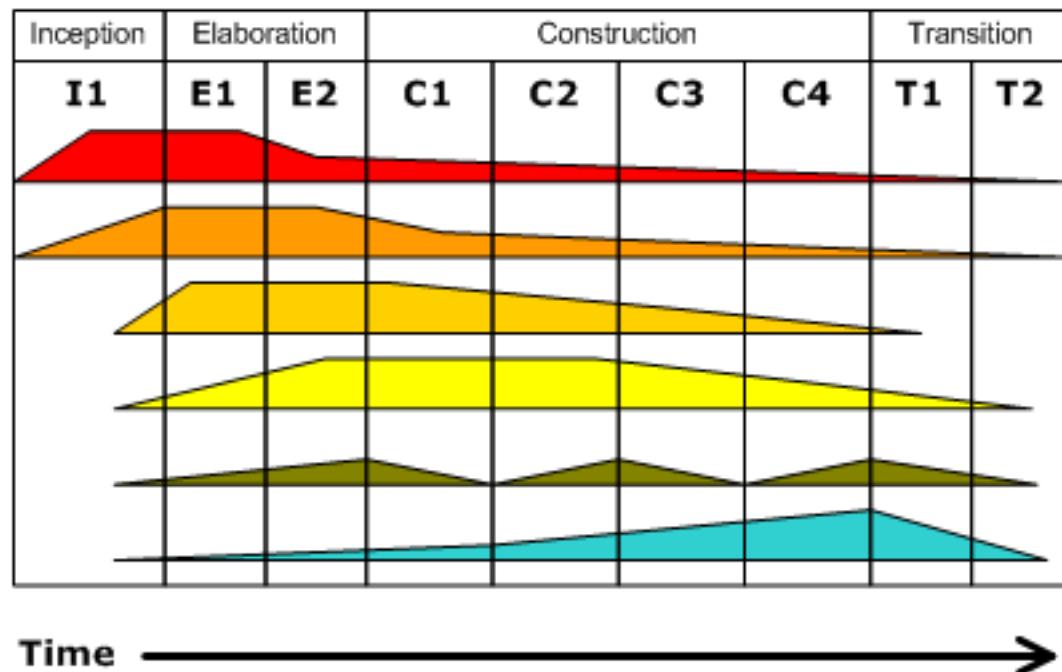
What is (Rational) Unified Process

- While UML provided the necessary technology for OO software design
- Unified process gives a framework to build the software using UML
- Iterative approach
- Five main phases
 - Inception
 - Establish a justification and define project scope
 - Outline the use cases and key requirements that will drive the design tradeoffs Outline one or more candidate architectures
 - Identify risks and prepare a preliminary project schedule alongwith cost estimate
 - Elaboration (Architecture and Design)
 - Construction (Actual implementation)
 - Transition (initial release)
 - Production (Actual deployment)

Architecture Activities

Iterative Development

Business value is delivered incrementally in time-boxed cross-discipline iterations.



- Architecture Focused all the time
- Starts during inception and described in detail during the elaboration phase

Feature Driven Design

- Starts during Inception-Elaboration Phase
- Mostly used in the context of Agile development
- The requirement is modeled as a set of features
 - A feature is a client-valued functionality that can be implemented and demonstrated quickly
- ‘Feature’ template **<action> the <result> (by|for|of|to) a(n) <object>**
 - Add a product to a shopping-cart
 - Store the shipping-information for the customer

From Feature to Architecture

- Once a set of features are collected
- Similar features are grouped together into a module
- Set of clusters created out of the features, forms a set of modules
- This becomes the basic Module Structure
 - Recall the software architecture and views...

Use-case Model

- Use-cases are used to describe an usage scenario from the user's point of view
 - Create a basic use-case diagram
 - Elaborate each use-case
- To create Use-Case
 - Define actors (who will use this use-case)
 - Describe the scenario
 - Preconditions
 - Main tasks
 - Exceptions
 - Variation in the actors interaction
 - What system information will the actor acquire, produce or change?
 - Will the actor have to inform about the changes?
 - Does the actor wish to be informed about any unexpected changes?

First step towards Module Views

- Analysis Classes are not the final implementation level classes. They are more coarse grained. They are typically modules.
They manifest as
 - External Entities
 - Other systems, devices that produce or consume information related to this system
 - Things
 - Report, letters, signals
 - Structure
 - Sensors, four-wheeled car,... that define a class of objects
-

Analysis Classes manifest as...

- Event Occurrences
 - Transfer of fund, Completion of Job
 - Roles
 - People who interact with the systems (Manager, engineer, etc.)-- Actors
 - Organizational Units
 - Divisions or groups that are important for this system
 - Places
 - Location that establish the context of the overall functionality of the system
-

How to get these classes?

- Get the noun phrases. They are the potential objects.
- Apply the following heuristics to get a legitimate analysis class
 - It should retain information for processing
 - It should have a set of identifiable operations
 - Multiple attributes should be there for a class
 - Operations should be applicable for all instances of this class
 - Attributes should be meaningful for all instances of this class

Thank You

Reference Chapter 18
Software Architecture in Practice
Third Edition
Len Bass
Paul Clements
Rick Kazman



BITS Pilani
Pilani|Dubai|Goa|Hyderabad

Software Architecture

Designing & Documenting the Architecture #2

Harvinder S Jabbal
Module RL7.0

Designing & Documenting the Architecture #2



Documenting the
Architecture with relevant
views

Module,

C&C,

Allocation &

Quality

Documenting beyond
views,

Documenting
behaviours with
sequence
diagram



Documenting the Architecture with relevant views – Module, C&C, Allocation & Quality

Architecture Documentation

Even the best architecture will be useless if the people who need it

- do not know what it is;
- cannot understand it well enough to use, build, or modify it;
- misunderstand it and apply it incorrectly.

All of the effort, analysis, hard work, and insightful design on the part of the architecture team will have been wasted.

Chapter Outline

1. Uses and Audiences for Architecture Documentation

2. Notations for Architecture Documentation

3. Views

4. Choosing the Views

5. Combining Views

6. Building the Documentation Package

7. Documenting Behavior

8. Architecture Documentation and Quality Attributes

9. Documenting Architectures That Change Faster Than You Can Document Them

10 Documenting Architecture in an Agile Development Project

11. Summary



1. Uses and Audience for Architecture Documentation

1. Uses and Audience for Architecture Documentation

Architecture documentation must

- be sufficiently transparent and accessible to be quickly understood by new employees
- be sufficiently concrete to serve as a blueprint for construction
- have enough information to serve as a basis for analysis.

Architecture documentation is both prescriptive and descriptive.

- For some audiences, it prescribes what *should* be true, placing constraints on decisions yet to be made.
- For other audiences, it describes what *is* true, recounting decisions already made about a system's design.

Understanding stakeholder uses of architecture documentation is essential

- Those uses determine the information to capture.

Three Uses for Architecture Documentation



Education

- Introducing people to the system
 - New members of the team
 - External analysts or evaluators
 - New architect

Primary vehicle for communication among stakeholders

- Especially architect to developers
- Especially architect to future architect!

Basis for system analysis and construction

- documentation serves as the basis for architecture evaluation.



2. Notations

2. Notations

Informal notations

- Views are depicted (often graphically) using general-purpose diagramming and editing tools
- The semantics of the description are characterized in natural language
- They cannot be formally analyzed

Semiformal notations

- Standardized notation that prescribes graphical elements and rules of construction
- Lacks a complete semantic treatment of the meaning of those elements
- Rudimentary analysis can be applied
- UML is a semiformal notation in this sense.

Formal notations

- Views are described in a notation that has a precise (usually mathematically based) semantics.
- Formal analysis of both syntax and semantics is possible.
- Architecture description languages (ADLs)
- Support automation through associated tools.

Choosing a Notation

Tradeoffs

- Typically, more formal notations take more time and effort to create and understand, but offer reduced ambiguity and more opportunities for analysis.
- Conversely, more informal notations are easier to create, but they provide fewer guarantees.

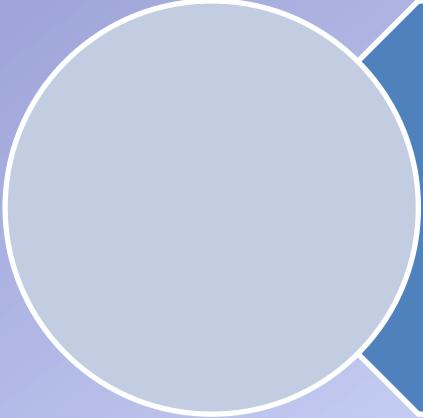
Different notations are better (or worse) for expressing different kinds of information.

- UML class diagram will not help you reason about schedulability, nor will a sequence chart tell you very much about the system's likelihood of being delivered on time.
- Choose your notations and representation languages knowing the important issues you need to capture and reason about.

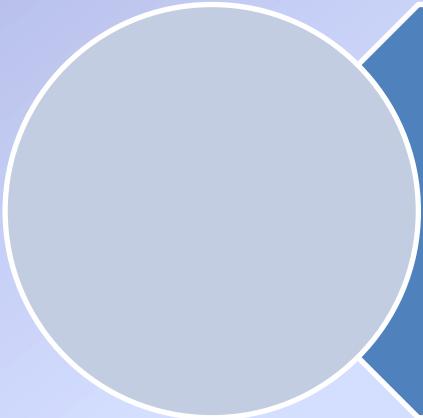


3. Views

3. Views



Views let us divide a software architecture into a number of (we hope) interesting and manageable representations of the system.



Principle of architecture documentation:

- *Documenting an architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view.*

Which Views? The Ones You Need!



Different views support different goals and uses.

We do not advocate a particular view or collection of views.

The views you should document depend on the uses you expect to make of the documentation.

Each view has a cost and a benefit; you should ensure that the benefits of maintaining a view outweigh its costs.

Overview of Module Views

Elements

- Modules, which are implementation units of software that provide a coherent set of responsibilities.

Relations

- *Is part of*, which defines a part/whole relationship between the submodule—the part—and the aggregate module—the whole.
- *Depends on*, which defines a dependency relationship between two modules. Specific module views elaborate what dependency is meant.
- *Is a*, which defines a generalization/specialization relationship between a more specific module—the child—and a more general module—the parent.

Overview of Module Views

Constraints

- Different module views may impose specific topological constraints, such as limitations on the visibility between modules.

Usage

- Blueprint for construction of the code
- Change-impact analysis
- Planning incremental development
- Requirements traceability analysis
- Communicating the functionality of a system and the structure of its code base
- Supporting the definition of work assignments, implementation schedules, and budget information
- Showing the structure of information that the system needs to manage

Module Views

- It is unlikely that the documentation of any software architecture can be complete without at least one module view.

Overview of C&C Views

Elements

- *Components*. Principal processing units and data stores. A component has a set of *ports* through which it interacts with other components (via connectors).
- *Connectors*. Pathways of interaction between components. Connectors have a set of roles (interfaces) that indicate how components may use a connector in interactions.

Relations

- *Attachments*. Component ports are associated with connector roles to yield a graph of components and connectors.
- *Interface delegation*. In some situations component ports are associated with one or more ports in an “internal” subarchitecture. The case is similar for the roles of a connector

Overview of C&C Views

Constraints

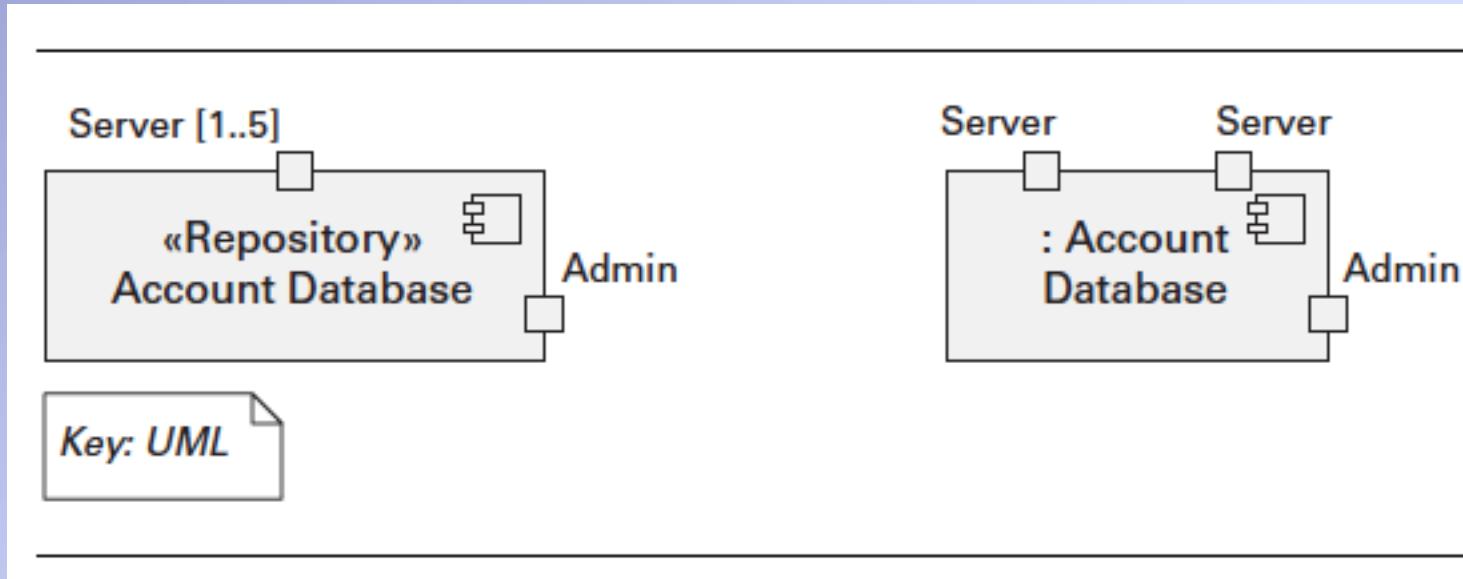
- Components can only be attached to connectors, not directly to other components.
- Connectors can only be attached to components, not directly to other connectors.
- Attachments can only be made between compatible ports and roles.
- Interface delegation can only be defined between two compatible ports (or two compatible roles).
- Connectors cannot appear in isolation; a connector must be attached to a component.

Usage

- Show how the system works.
- Guide development by specifying structure and behavior of runtime elements.
- Help reason about runtime system quality attributes, such as performance and availability.

Notations for C&C Views

UML components are good match for C&C components.



Suggested Reading:

<http://agilemodeling.com/artifacts/componentDiagram.htm>

Notations for C&C Views

UML connectors are not rich enough to represent many C&C connectors.

- UML connectors cannot have substructure, attributes, or behavioral descriptions.

Represent a “simple” C&C connector using a UML connector—a line.

- Many commonly used C&C connectors have well-known, application-independent semantics and implementations, such as function calls or data read operations.
- You can use a stereotype to denote the type of connector.

Connector roles cannot be explicitly represented with a UML connector.

- The UML connector element does not allow the inclusion of interfaces.
- Label the connector ends and use these labels to identify role descriptions that must be documented elsewhere.

Represent a “rich” C&C connector

- using a UML component, or by annotating a line UML connector with a tag that explains the meaning of the complex connector.

Overview of Allocation Views

Elements

- *Software element* and *environmental element*.
- A software element has properties that are *required* of the environment.
- An environmental element has properties that are *provided* to the software.

Relations

- *Allocated to*. A software element is mapped (allocated to) an environmental element. Properties are dependent on the particular view.

Overview of Allocation Views

Constraints

- Varies by view

Usage

- Reasoning about performance, availability, security, and safety.
- Reasoning about distributed development and allocation of work to teams.
- Reasoning about concurrent access to software versions.
- Reasoning about the form and mechanisms of system installation.

Quality Views

A *quality view* can be tailored

- for specific stakeholders or to address specific concerns.

A quality views is formed

- by extracting the relevant pieces of structural views and packaging them together.

Quality Views: Examples

- Show the components that have some security role or responsibility, how those components communicate, any data repositories for security information, and repositories that are of security interest.
- The view's context information would show other security measures (such as physical security) in the system's environment.
- The behavior part of a security view
 - Show how the operation of security protocols and where and how humans interact with the security elements.
 - Capture how the system would respond to

Security view

- Especially helpful for systems that are globally dispersed and heterogeneous.
- Show all of the component-to-component channels, the various network channels, quality-of-service parameter values, and areas of concurrency.
- Used to analyze certain kinds of performance and reliability (such as deadlock or race condition detection).
- The behavior part of this view could show (for example) how network bandwidth is dynamically allocated.

Communications view

Quality Views: Examples

- Could help illuminate and draw attention to error reporting and resolution mechanisms.
- Show how components detect, report, and resolve faults or errors.
- It would help identify the sources of errors

*Exception or
error-handling
view*

- Models mechanisms such as replication and switchover.
- Depicts timing issues and transaction integrity.

Reliability view

- Shows those aspects of the architecture useful for inferring the system's performance.
- Show network traffic models, maximum latencies for operations, and so forth.

*Performance
view*



Sample separator slide for the presentation

4. Choosing the Views

You can determine which views are required, when to create them, and how much detail to include if you know the following:

What people, and with what skills, are available	Which standards you have to comply with	What budget is on hand	What the schedule is	What the information needs of the important stakeholders are	What the driving quality attribute requirements are	What the approximate size of the system is
--	---	------------------------	----------------------	--	---	--



5. Combining Views

5. Combining Views

At a minimum, expect to have

at least one module view,

at least one C&C view,

and for larger systems, at least one allocation view in your architecture document.

Method for Choosing the Views



Step 1. Build a stakeholder/view table.

- Rows: List the stakeholders for your project's software architecture documentation
- Columns: Enumerate the views that apply to your system.
 - Use the structures discussed in Chapter 1, the views discussed in this chapter, and the views that your design work in ADD has suggested as a starting list of candidates.
 - Include the views or view sketches you have as a result of your design work so far.
- Some views (such as decomposition, uses, and work assignment) apply to every system, while others (various C&C views, the layered view) only apply to some systems.
- Fill in each cell to describe how much information the stakeholder requires from the view: none, overview only, moderate detail, or high detail.

Method for Choosing the Views

innovate

achieve

lead

Step 2.
Combine
views to
reduce
their
number

- Look for marginal views in the table; those that require only an overview, or that serve very few stakeholders.
- Combine each marginal view with another view that has a stronger constituency.
- These views often combine naturally:
 - *Various C&C views.* Because C&C views all show runtime relations among components and connectors of various types, they tend to combine well.
 - *Deployment view with either SOA or communicating-processes views.* An SOA view shows services, and a communicating-processes view shows processes. In both cases, these are components that are deployed onto processors.
 - *Decomposition view and any of work assignment, implementation, uses, or layered views.* The decomposed modules form the units of work, development, and uses. In addition, these modules populate layers.

Method for Choosing the Views



Step 3. Prioritize and stage.

- The decomposition view (one of the module views) is a particularly helpful view to release early.
 - High-level decompositions are often easy to design
 - The project manager can start to staff development teams, put training in place, determine which parts to outsource, and start producing budgets and schedules.
- You don't have to satisfy all the information needs of all the stakeholders to the fullest extent.
 - Providing 80 percent of the information goes a long way, and this might be "good enough" so that the stakeholders can do their job.
 - Check with the stakeholder if a subset of information would be sufficient.
- You don't have to complete one view before starting another.
 - People can make progress with overview-level information
 - A breadth-first approach is often the best.



6. Building the Documentation Package

6. Building the Documentation Package



Documentation package consists of

Views

Documentation beyond views

Documenting a View

Section 1: The Primary Presentation.

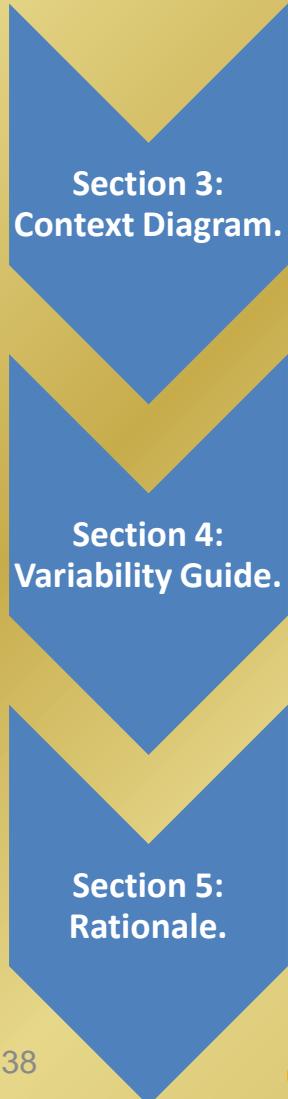
- The *primary presentation* shows the elements and relations of the view.
- The primary presentation should contain the information you wish to convey about the system—in the vocabulary of that view.
- The primary presentation is most often graphical.
 - It might be a diagram you've drawn in an informal notation using a simple drawing tool, or it might be a diagram in a semiformal or formal notation imported from a design or modeling tool that you're using.
 - If your primary presentation is graphical, make sure to include a key that explains the notation.
 - Lack of a key is the most common mistake that we see in documentation in practice.
- Occasionally the primary presentation will be textual, such as a table or a list.
 - If that text is presented according to certain stylistic rules, these rules should be stated or incorporated by reference, as the analog to the graphical notation key.

Documenting a View

Section 2: The Element Catalog.

- The *element catalog* details at least those elements depicted in the primary presentation.
 - For instance, if a diagram shows elements A, B, and C, then the element catalog needs to explain what A, B, and C are.
 - If elements or relations relevant to this view were omitted from the primary presentation, they should be introduced and explained in the catalog.
- Parts of the catalog:
 - *Elements and their properties*. This section names each element in the view and lists the properties of that element. Each view introduced in Chapter 1 listed a set of suggested properties associated with that view.
 - *Relations and their properties*. Each view has specific relation types that it depicts among the elements in that view.
 - *Element interfaces*. This section documents element interfaces.
 - *Element behavior*. This section documents element behavior that is not obvious from the primary presentation.

Documenting a View

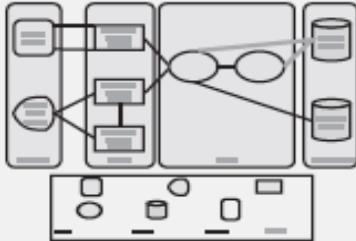


- A *context diagram* shows how the system or portion of the system depicted in this view relates to its environment.
 - The purpose of a context diagram is to depict the scope of a view.
 - Entities in the environment may be humans, other computer systems, or physical objects, such as sensors or controlled devices.
-
- A *variability guide* shows how to exercise any variation points that are a part of the architecture shown in this view.
-
- *Rationale* explains why the design reflected in the view came to be.
 - The goal of this section is to explain why the design is as it is and to provide a convincing argument that it is sound.
 - The choice of a pattern in this view should be justified here by describing the architectural problem that the chosen pattern solves and the rationale for choosing it over another.

View Template

Template for a View

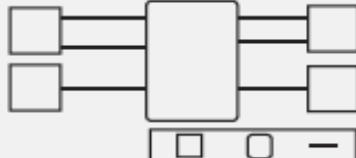
Section 1. Primary Presentation



Section 2. Element Catalog

- Section 2.A. Elements and Their Properties
- Section 2.B. Relations and Their Properties
- Section 2.C. Element Interfaces
- Section 2.D. Element Behavior

Section 3. Context Diagram



Section 4. Variability Guide

Section 5. Rationale



Documenting beyond views,

Documenting Information Beyond Views



Document control information.

List the

- issuing organization,
- the current version number,
- date of issue and
- status,
- a change history, and
- the procedure for submitting change requests to the document.

Usually captured in the front matter

Documenting Information Beyond Views



Section 1: Documentation

Roadmap. The documentation map tells the reader what information is in the documentation and where to find it.

- *Scope and summary.* Explain the purpose of the document and briefly summarize what is covered.
- *How the documentation is organized.* For each section in the documentation, give a short synopsis of the information that can be found there.
- *View overview.* Describes the views that the architect has included in the package. For each view::
 - The name of the view and what pattern it instantiates, if any.
 - A description of the view's element types, relation types, and property types.
 - A description of language, modeling techniques, or analytical methods used in constructing the view.
- *How stakeholders can use the documentation.*
 - This section shows how various stakeholders might use the documentation to help address their concerns.
 - Include short scenarios, such as “A maintainer wishes to know the units of software that are likely to be changed by a proposed modification.”
 - To be compliant with ISO/IEC 42010-2007, you must consider the concerns of at least users, acquirers, developers, and maintainers.

Documenting Information Beyond Views



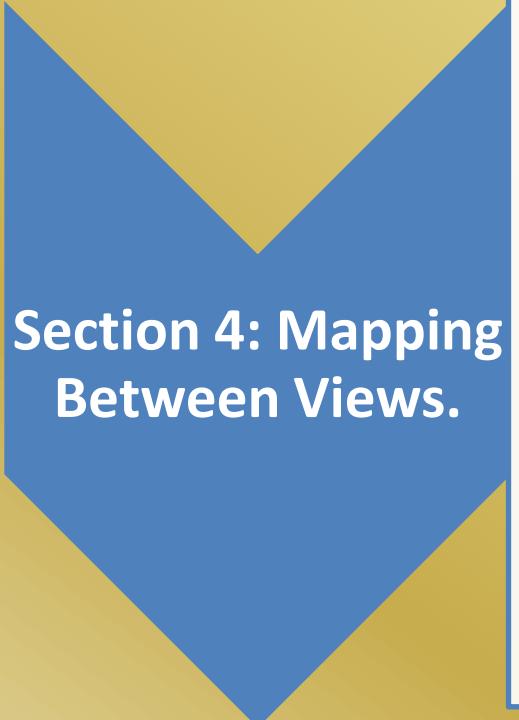
Section 2: How a View Is Documented.

- Explain the standard organization you're using to document views—either the one described in this chapter or one of your own.

Section 3: System Overview.

- Short prose description of the system's function, its users, and any important background or constraints.
- Provides your readers with a consistent mental model of the system and its purpose.
- This might be a pointer to your project's concept-of-operations document for the system.

Documenting Information Beyond Views



Section 4: Mapping Between Views.

- Helping a reader understand the associations between views will help that reader gain a powerful insight into how the architecture works as a unified conceptual whole.
- The associations between elements across views in an architecture are, in general, many-to-many.
- View-to-view associations can be captured as tables.
 - The table should name the correspondence between the elements across the two views.
 - Examples
 - “is implemented by” for mapping from a component-and-connector view to a module view
 - “implements” for mapping from a module view to a component-and-connector view
 - “included in” for mapping from a decomposition view to a layered view

Documenting Information Beyond Views



Section 5: Rationale.

- Documents the architectural decisions that apply to more than one view.
 - Documentation of background or organizational constraints or major requirements that led to decisions of system-wide import.
 - Decisions about which fundamental architecture patterns are used.

Section 6: Directory.

- Set of reference material that helps readers find more information quickly.
 - Index of terms
 - Glossary
 - Acronym list.



7 Documenting behaviours

7. Documenting Behavior

Behavior documentation complements each views by describing how architecture elements in that view interact with each other.

Behavior documentation enables reasoning about

- a system's potential to deadlock
- a system's ability to complete a task in the desired amount of time
- maximum memory consumption
- and more

Behavior has its own section in our view template's element catalog.

Notations for Documenting Behavior



Trace-oriented languages

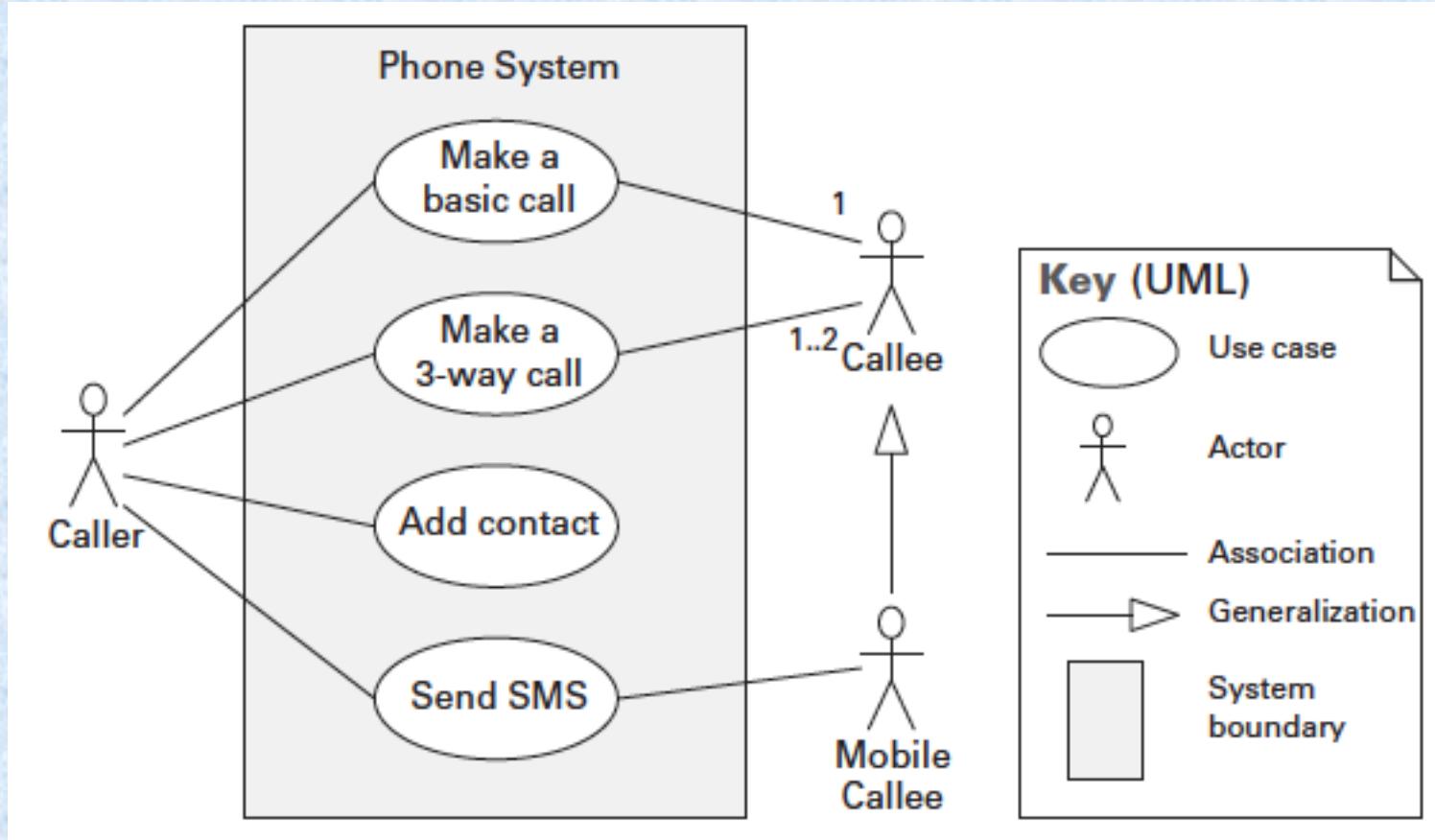
Traces are sequences of activities or interactions that describe the system's response to a specific stimulus when the system is in a specific state.

A trace describes a particular sequence of activities or interactions between structural elements of the system.

Examples

- use cases
- sequence diagrams
- communication diagrams
- activity diagrams
- message sequence charts
- timing diagrams
- Business Process Execution Language

Use Case Diagram



Use Case Description

Name: Make a basic call

Description: Making a point-to-point connection between two phones.

Primary actors: Caller

Secondary actors: Callee

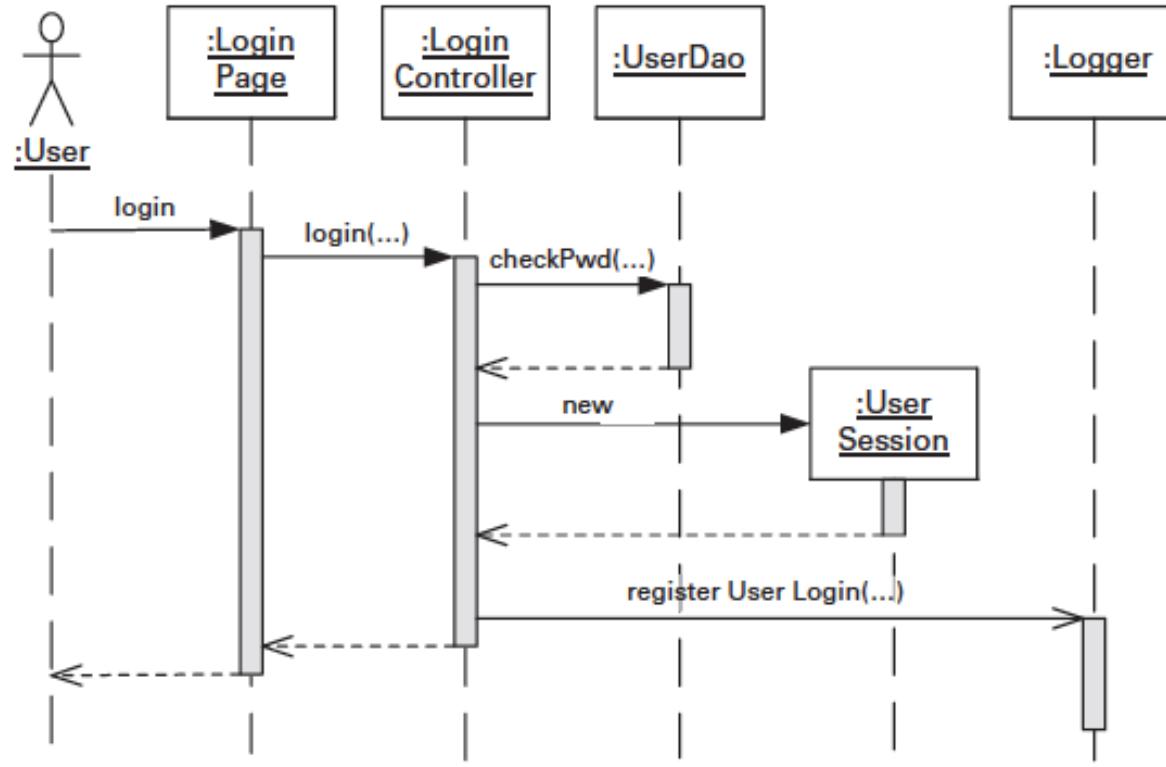
Flow of events:

The use case starts when a caller places a call via a terminal, such as a cell phone. All terminals to which the call should be routed then begin ringing. When one of the terminals is answered, all others stop ringing and a connection is made between the caller's terminal and the terminal that was answered. When either terminal is disconnected—someone hangs up—the other terminal is also disconnected. The call is now terminated, and the use case is ended.

Exceptional flow of events:

The caller can disconnect, or hang up, before any of the ringing terminals has been answered. If this happens, all ringing terminals stop ringing and are disconnected, ending the use case.

Sequence Diagram



Key (UML)



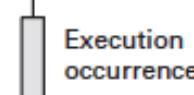
Actor



Object



Lifeline



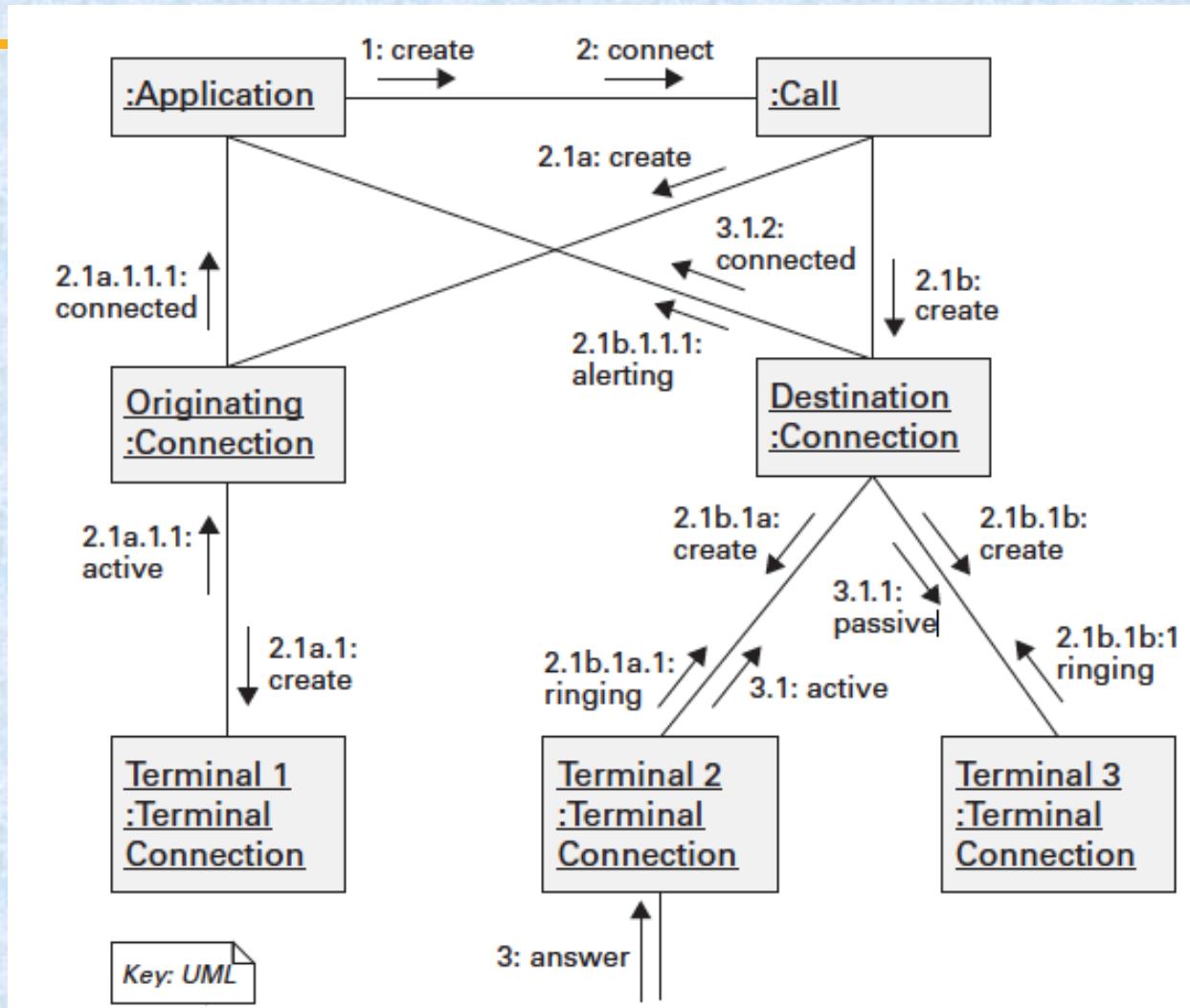
Execution occurrence

→ Synchronous message

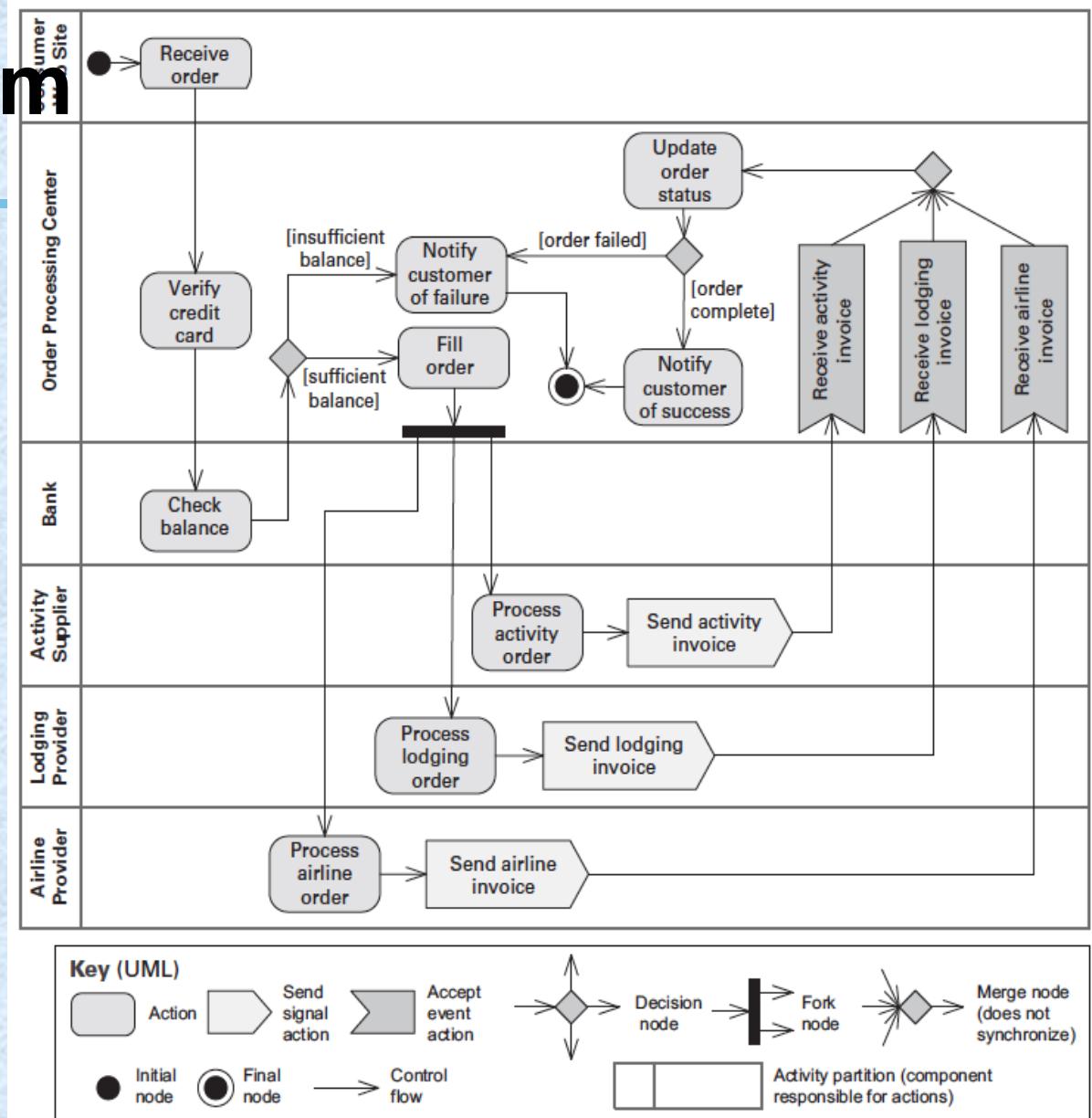
→ Asynchronous message

→ Return message

Communication Diagram



Activity Diagram

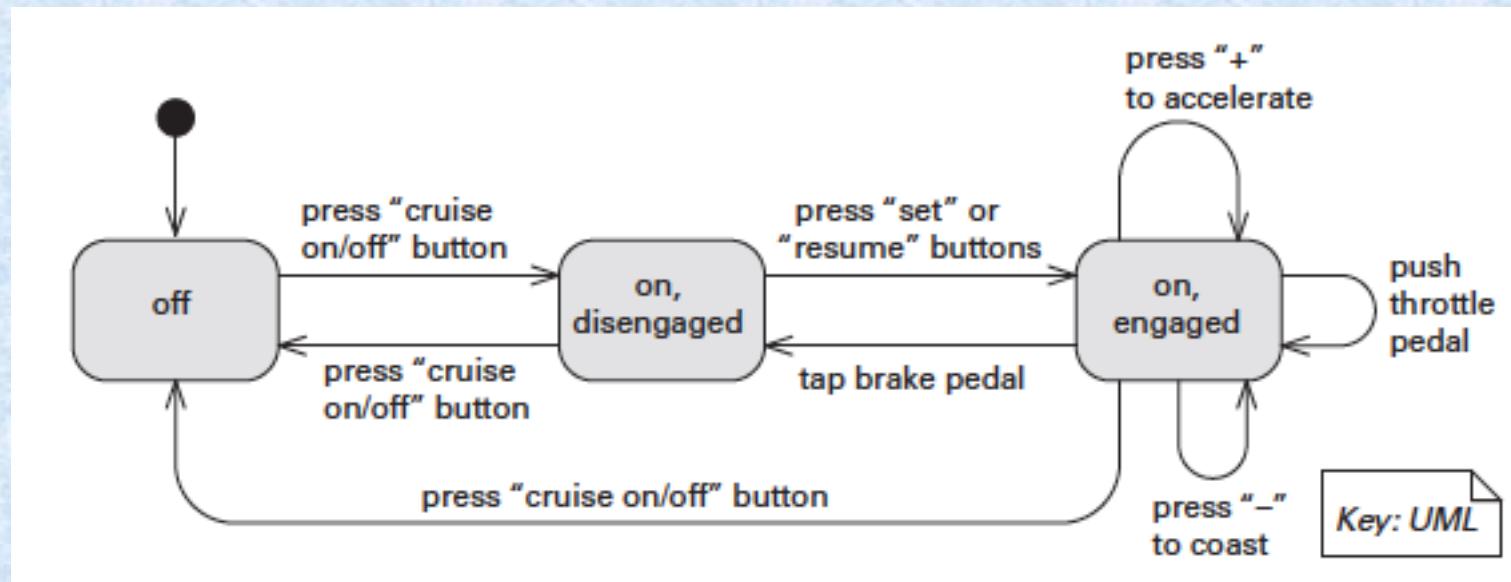


Notations for Documenting Behavior

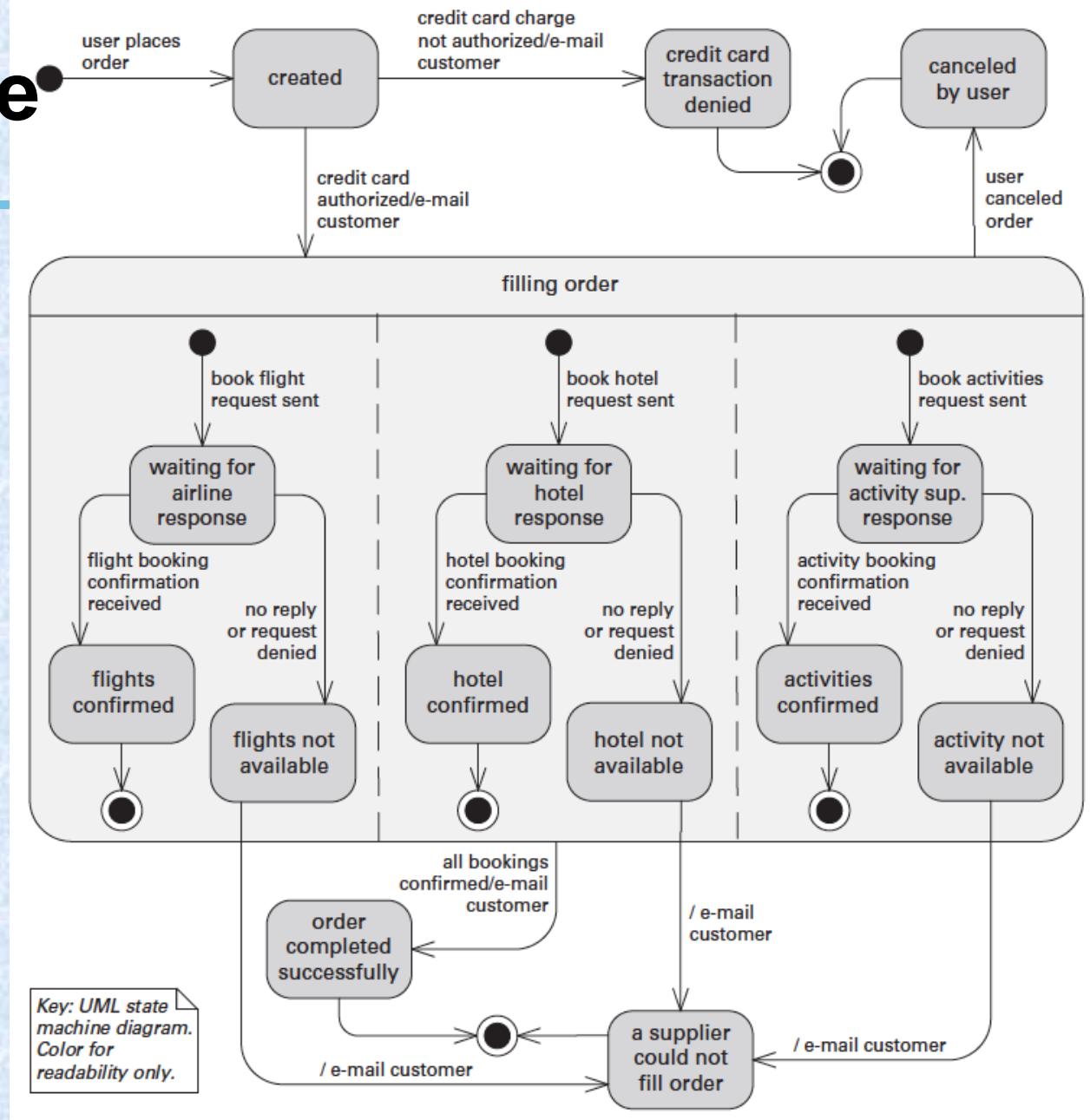
Comprehensive languages

- *Comprehensive models* show the complete behavior of structural elements.
- Given this type of documentation, it is possible to infer all possible paths from initial state to final state.
- The state machine formalism represents the behavior of architecture elements because each state is an abstraction of all possible histories that could lead to that state.
- State machine languages allow you to complement a structural description of the elements of the system with constraints on interactions and timed reactions to both internal and environmental stimuli.

State Machine



State Machine





8. Documenting Quality Attributes

8. Documenting Quality Attributes



Where do quality attributes show up in the documentation? There are five major ways:

Rationale that explains the choice of design approach should include a discussion about the quality attribute requirements and tradeoffs.	Architectural elements providing a service often have quality attribute bounds assigned to them, defined in the interface documentation for the elements, or recorded as <i>properties</i> that the elements exhibit.	Quality attributes often impart a “language” of things that you would look for. Someone fluent in the “language” of a quality attribute can search for the kinds of architectural elements put in place to satisfy that quality attribute requirement.	Architecture documentation often contains a <i>mapping to requirements</i> that shows how requirements (including quality attribute requirements) are satisfied.	Every quality attribute requirement will have a constituency of stakeholders who want to know that it is going to be satisfied. For these stakeholders, the roadmap tells the stakeholder where in the document to find it.
---	---	--	--	---



9. Documenting Architectures That Change Faster Than You Can Document Them

9. Documenting Architectures That Change Faster Than You Can Document Them



An architecture that changes at runtime, or as a result of a high-frequency release-and-deploy cycle, change much faster than the documentation cycle.

Nobody will wait until a new architecture document is produced, reviewed, and released.

In this case:

- *Document what is true about all versions of your system.* Record those invariants as you would for any architecture. This may make your documented architecture more a description of constraints or guidelines that any compliant version of the system must follow.
- *Document the ways the architecture is allowed to change.* This will usually mean adding new components and replacing components with new implementations. The place to do this is called the variability guide.



10. Documenting Architecture in an Agile Development Project

10. Documenting Architecture in an Agile Development Project



Adopt a template or standard organization to capture your design decisions.

Plan to document a view if (but only if) it has a strongly identified stakeholder constituency.

Fill in the sections of the template for a view, and for information beyond views, when (and in whatever order) the information becomes available. But only do this if writing down this information will make it easier (or cheaper or make success more likely) for someone downstream doing their job.

Don't worry about creating an architectural design document and then a finer-grained design document. Produce just enough design information to allow you to move on to code.

Don't feel obliged to fill up all sections of the template, and certainly not all at once. Write "N/A" for the sections for which you don't need to record the information (perhaps because you will convey it orally).

Agile teams sometimes make models in brief discussions by the whiteboard. Take a picture and use it as the primary presentation.

Summary

- You must understand the uses to which the writing is to be put and the audience for the writing.
- Architectural documentation serves as a means for communication among various stakeholders, not only up the management chain and down to the developers but also across to peers.
- An architecture is a complicated artifact, best expressed by focusing on views.
- You must choose the views to document, must choose the notation to document these views, and must choose a set of views that is both minimal and adequate.
- You must document not only the structure of the architecture but also the behavior.



Thank you.....

Credits

- **Chapter Reference from Text T1: 16, 17, 18**
- Slides have been adapted from Authors Slides
Software Architecture in Practice – Third Ed.
 - Len Bass
 - Paul Clements
 - Rick Kazman