SS ZG653 (RL 3.1): Software Architecture

Quality classes and attribute, quality attribute scenario and architectural tactics

Instructor: Prof. SantonuSarkar

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Jan 11, 2015

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

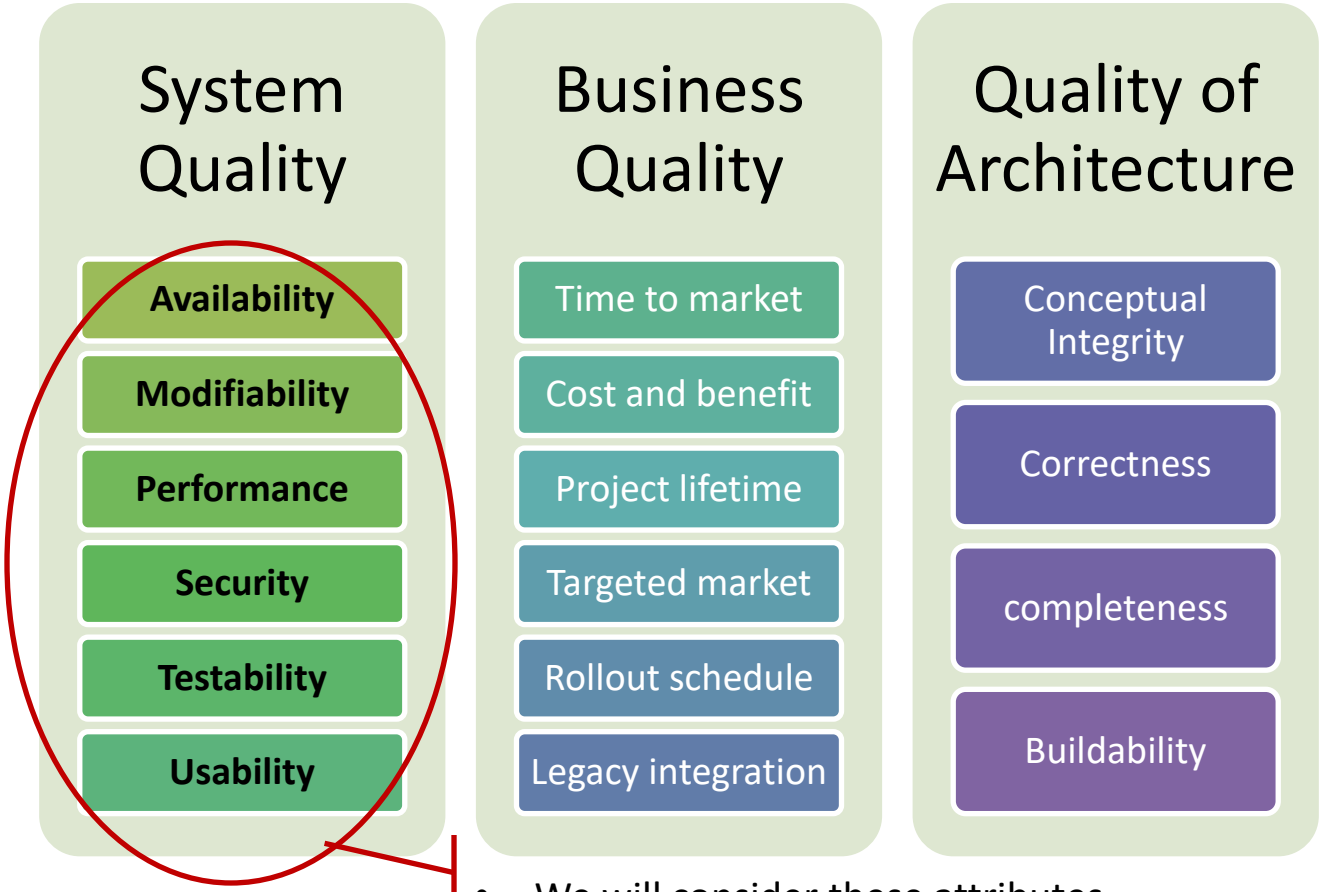
- Needs to be achieved throughout the design, implementation and deployment
- Should be designed in and also evaluated at the architectural level
- Quality attributes are NON-orthogonal
  - One can have an effect (positive or negative) on another
  - Performance is troubled by nearly all other. All other demand more code where-as 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

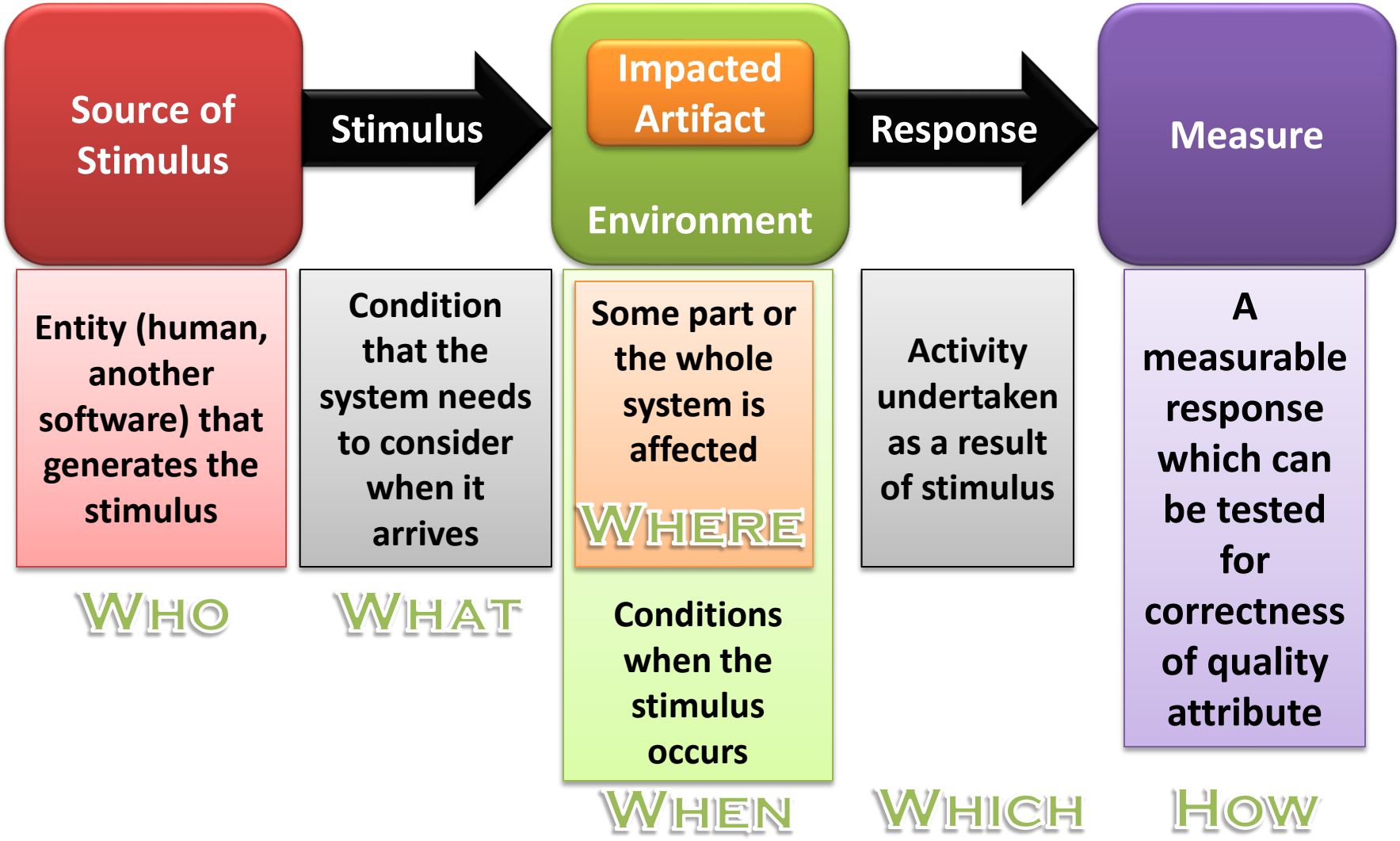
- ISO 9126 and ISO 25000 attempts to create a framework to define quality attributes

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	•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	•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	•The product thatneeds 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	•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	•Phased rollouts; base + additional features spaced in time •Flexibility and customizability become the key
Integration with Legacy System	•Appropriate integration mechanisms •Much implications on architecture

Architectural Qualities

Architectural Quality	Details
Conceptual Integrity	•Architecture should do similar things in similar ways •Unify the design at all levels
Correctness and Completeness	•Essential to ensure system's requirements and run time constraints are met
Build ability	•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

Jan 11, 2015

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

WHO

End user

STIMULUS

User Wants to
- Learn system feature
- Use systems efficiently
- Minimize the impact of errors
- Adapt system
- Feel comfortable

IMPACTED PART

Whole System

- **At run time**
- **At configure time**

MITIGATING ACTION
- Learn
  - ✓ Context sensitive help, familiar interface
- Efficient use
  - ✓ Aggregation of data and command, reuse of already entered data, good navigation, search mechanism, multiple activities
- Error impact
  - ✓ Undo, cancel, recover, auto-correct, retrieve forgotten information

MEASURABLE RESPONSE
- Task time
- Number of errors
- User satisfaction
- Gain of user knowledge
- Successful operations
- Amount of time/data lost

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)
  - Cancel, undo, aggregation, store partial result
- System initiative
  - 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

innovate achieve lead

Thank You

SS ZG653 (RL 4.2): Software Architecture

Availability and Its Tactics

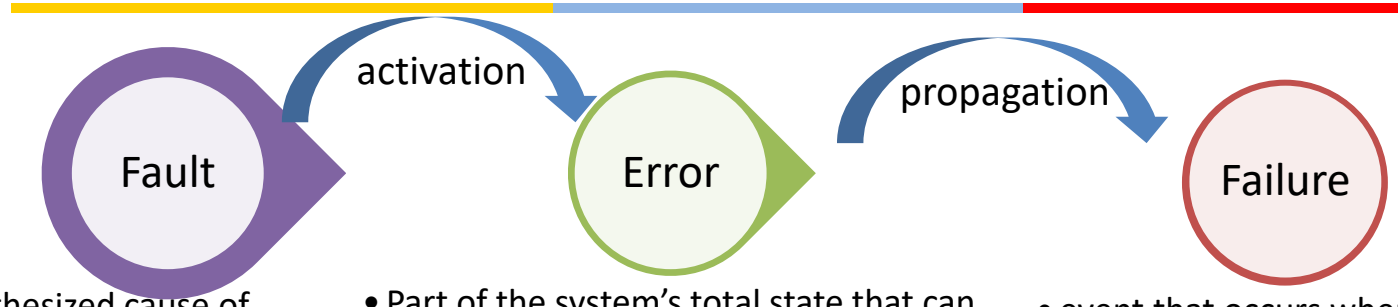
Instructor: Prof. Santonu Sarkar

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Jan 20, 2015

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

WHO

Internal or External to System

STIMULUS

Fault causing
- System does not respond
- Crash
- Delay in response
- Errorneous Response

IMPACTED PART

Infrastructure and/or application

- During normal operation
- During degraded mode of operation

MITIGATING ACTION

When fault occurs it should do one or more of
- ✓ 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

MEASURABLE RESPONSE

- 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
• Ping/echo • Heartbeat • Timestamp • Data sanity check • Condition monitoring • Voting • Exception Detection • Self-test	• Active redundancy (Hot) • Passive redundancy (Warm) • Spare (Cold) • Exception handling • Graceful degradation • Ignore faulty behavior	• Rollback • Retry • Reconfiguration • Shadow operation • State resynchronization • Escalating restart • Nonstop forwarding	• 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 an emergent system property

THANK YOU

SS ZG653 (RL 5.1): Software Architecture

Modifiability and Its Tactics

Instructor: Prof. Santonu Sarkar

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

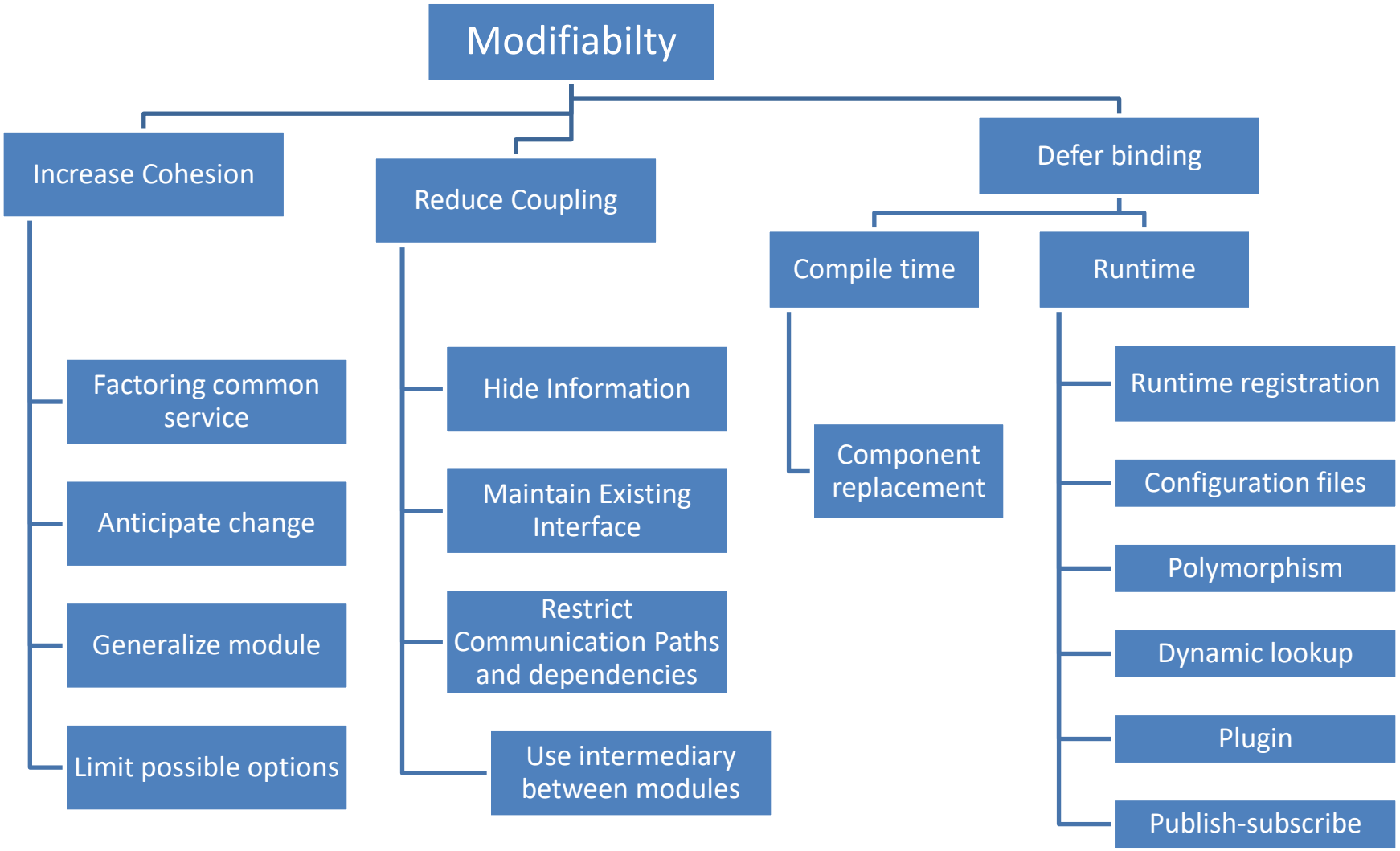
Modifiability

- **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 (cost of modification w/o modifiability mechanism in place) > (cost of modification with modifiability in place)+ Cost of installing the mechanism**

Modifiability Scenarios

WHO	STIMULUS	IMPACTED PART	MITIGATING ACTION	MEASURABLE RESPONSE
• Enduser • Developer • SysAdm	They want to modify ➢ Functionality ➢ Add, modify, delete ➢ Quality ➢ Capacity	UI, platform or System •Runtime •Compile time •Design time •Build time	When fault occurs it should do one or more of ✓ Locate (Impact analysis) ✓ Modify ✓ Test ✓ Deploy again	• 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

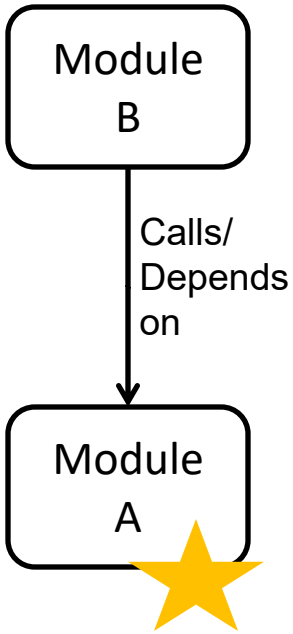
- B may assume that A must exist when B is calling A

Resource behavior of A

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

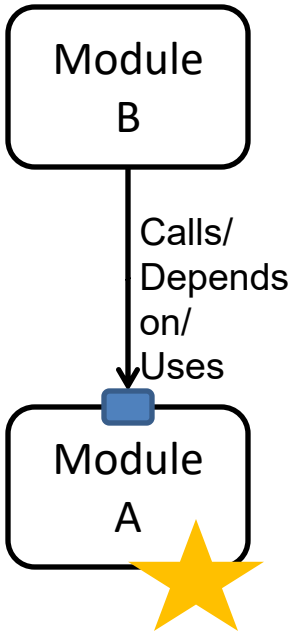
Localize Modifications

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



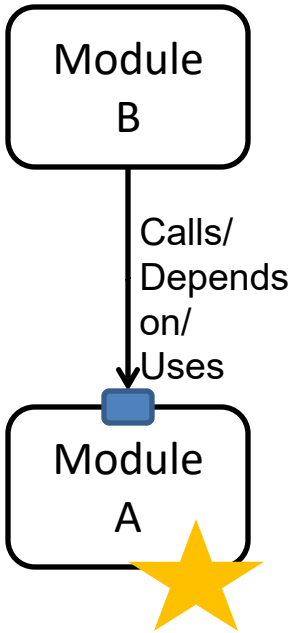
Prevent Ripple Effect Tactics

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

SS ZG653 (RL 5.2): Software Architecture

Performance and Its Tactics

Instructor: Prof. Santonu Sarkar

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

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

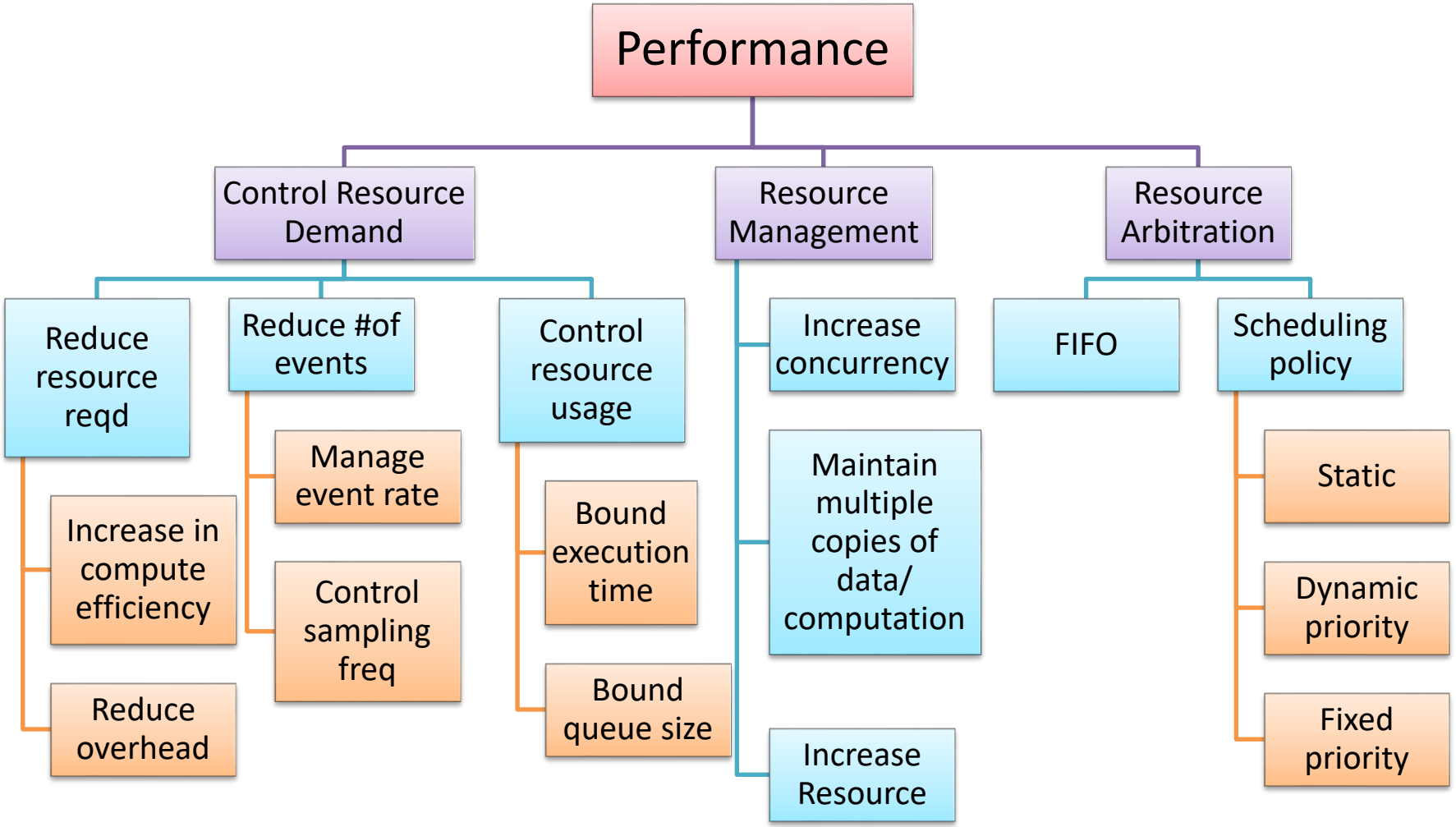
WHO	STIMULUS	IMPACTED PART	MITIGATING ACTION	MEASURABLE RESPONSE
• External • Internal	•**Events that are** •Periodic •Sporadic •Stochastic (with avg 1000 txn/min)	System •Running in normal mode •Overloaded	✓ Processes event in time ✓ Changes the service-level	• Latency or deadline by which it must be processed • Throughput- #of events processed/time • Jitter- variation • Miss rate • Data loss
Users	Submits transactions and expect response in 3s		Transactions processed	Average latency of 2s

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 priorization, 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.

Performance Design Checklist- 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?

innovate achieve lead

Thank You

SS ZG653 (RL 6.1): Software Architecture

Security and Its Tactics

Instructor: Prof. Santonu Sarkar

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

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.

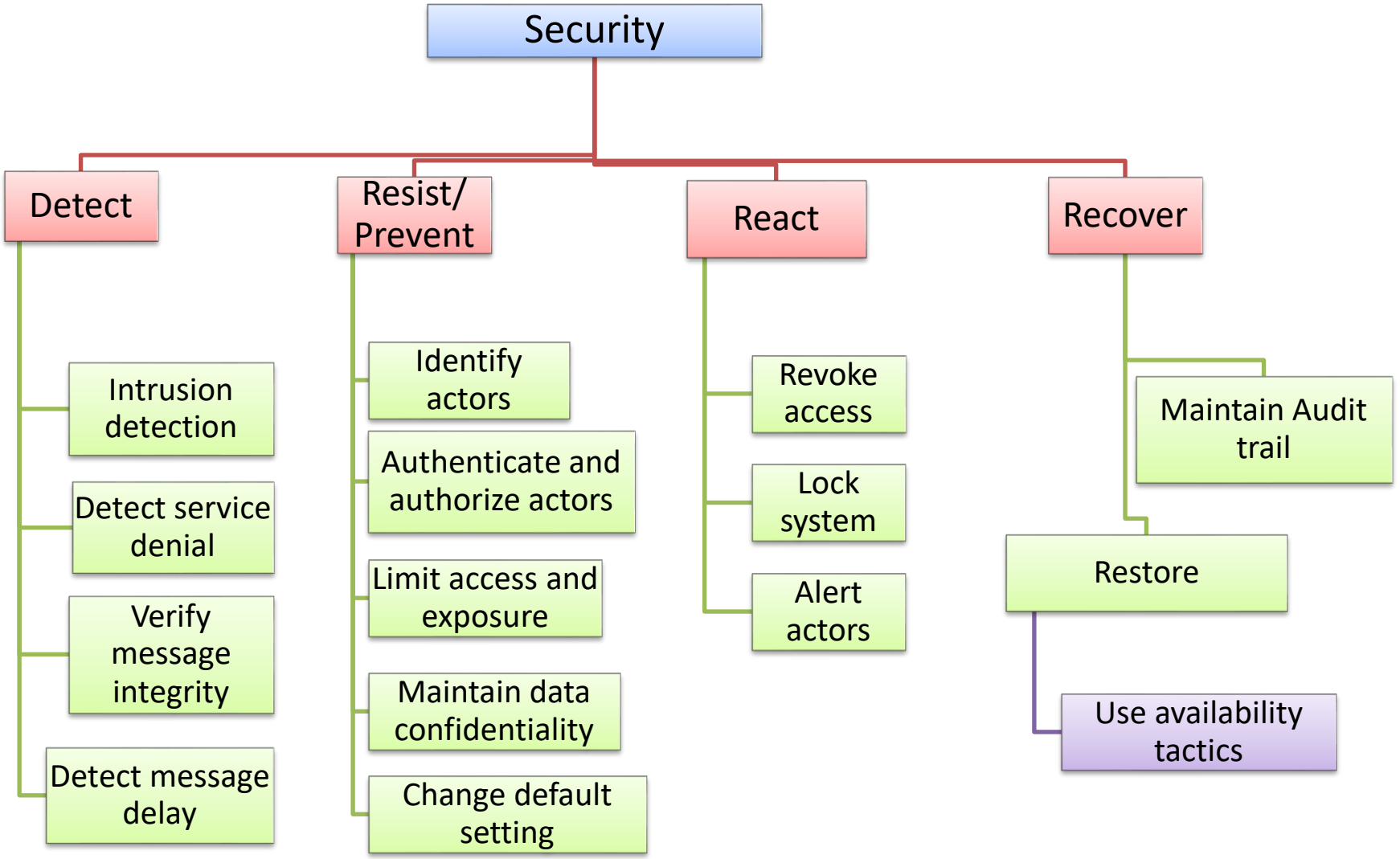
Security Scenario

WHO	STIMULUS	IMPACTED PART	MITIGATING ACTION	MEASURABLE RESPONSE
• **Individual or a system** ✓Correctly identified OR ✓Incorrectly identified ✓OR unknown • **Who is** ✓Internal or external ✓Authorized or unauthrized • **Has access to** ✓Limited resource ✓Vast resource	•**Tries to** ✓Read data ✓Change/delete data ✓Access system services ✓Reduce availability of services	System services, data •Online/offline •Within firewall or Open	•**Authentication** ✓Authenticates ✓Hides identity •**Access control to data or services** ✓Blocks access ✓Grants/withdraws permission to access ✓Audits access/ modification attempts •**Corrective Actions** ✓Encrypts data ✓Detect anomalous situation (high access req.) and informs people/another system ✓Restricts availability	• 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
An insider	Updates payment details	Pay database table	Anomaly detected Audit trail maintained	Corrective action taken in 1 day

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 *cominginto* 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

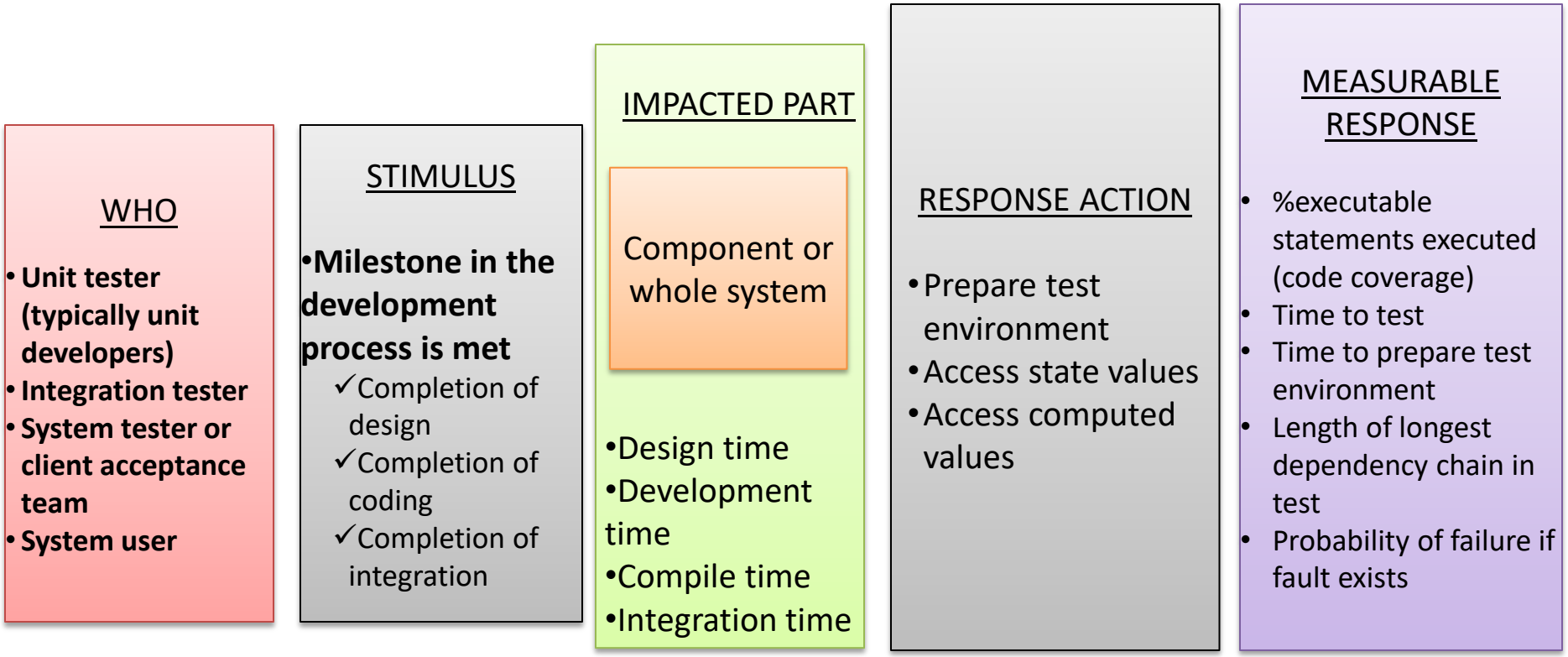
Dijkstra's Thesis

- 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

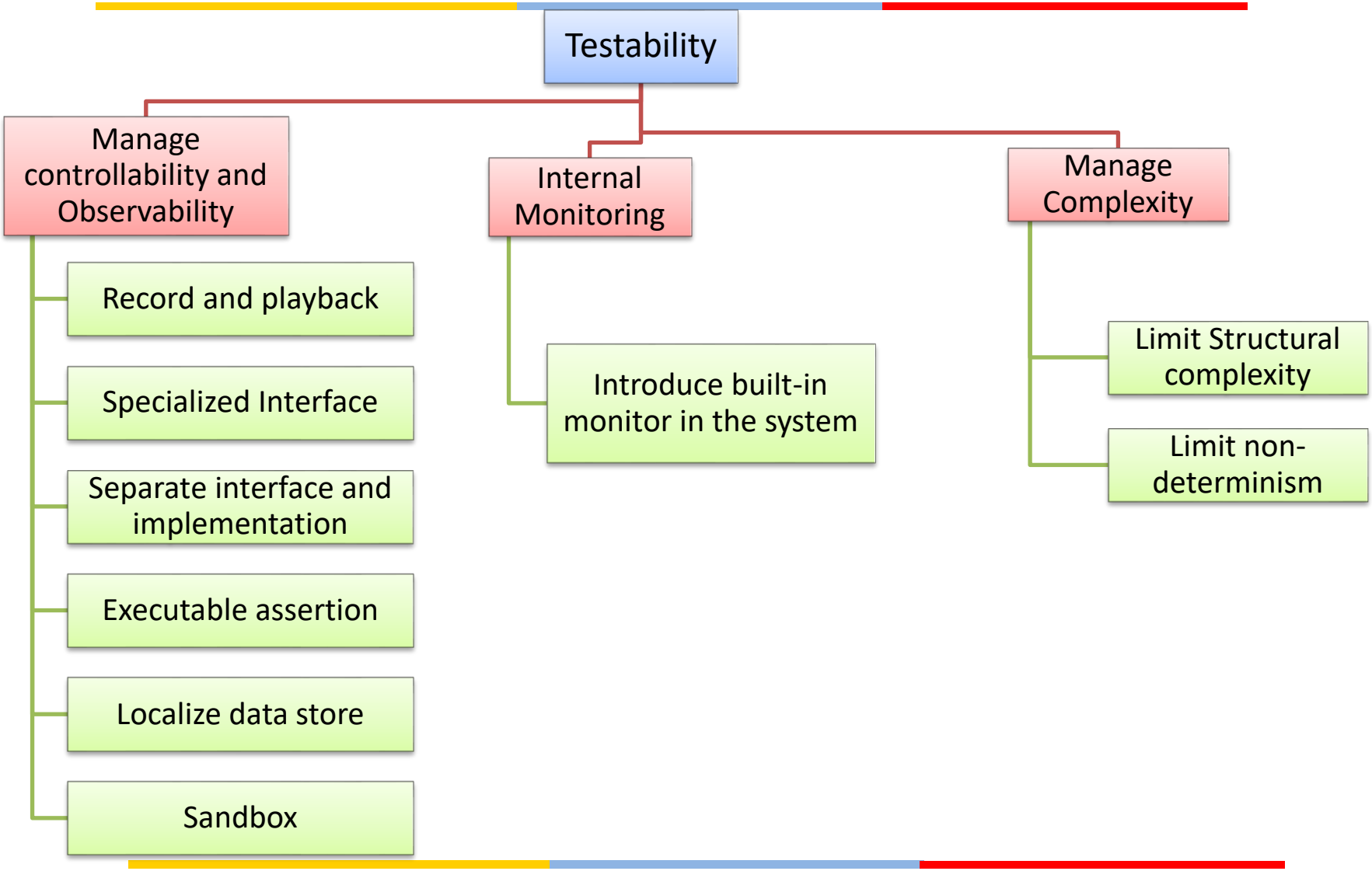


A unit tester	Performs unit test	Component that has controllable interface After component is complete	Observe the output for inputs provided	Coverage of 85% is achieved in 2 hours

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 toexplicitly 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

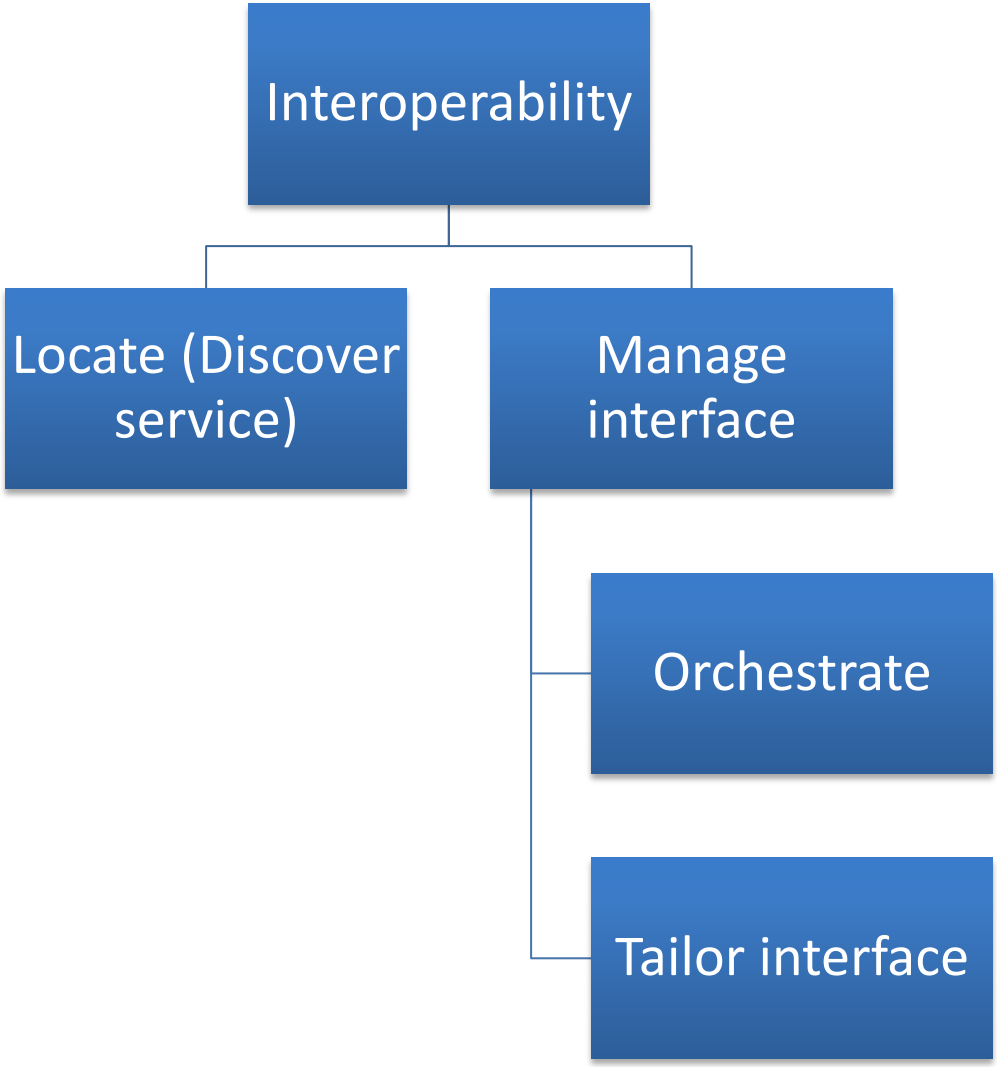
Interoperability Scenario

WHO	STIMULUS	IMPACTED PART	RESPONSE ACTION	MEASURABLE RESPONSE
• **A system initiating request**	• **Request to exchange information**	Component or whole system • It has been already discovered • at runtime • Prior to runtime	• The request is ACCEPTED and information is exchanged • The request is REJECTED and appropriate entities are notified • The request is logged by one or more systems	• %information that has been exchanged successfully • %information that has been REJECTED successfully
Avehicle information system	Sends current location	Traffic monitoring system -- discovered prior to runtime	Combines the location information with other details, overlays with Google Maps, and broadcasts	The vehicle information is included correctly 99.9% of the time

Notion of Interface

- Information exchange
  - Can be as simple as A calling B
  - A and B can exchange implicitly w/o direct communication
  - Operation Dessert 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
Locate (Discover service)
Manage interface
Orchestrate
Tailor interface

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. REST*ful* 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