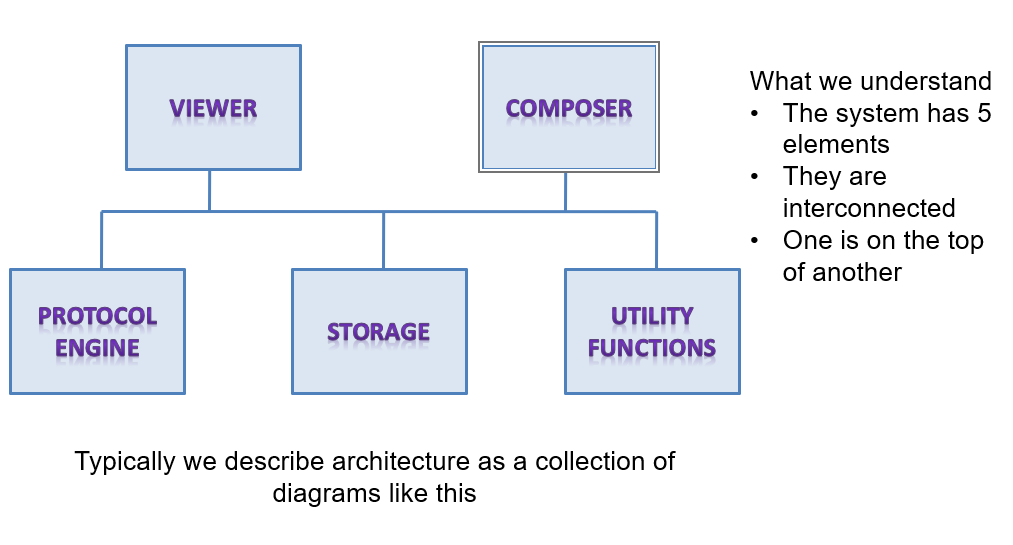
**Software Architecture**

**What is Software Architecture: --**

*The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.*

There is no good or bad architecture. Architecture always evolves. It has good rule of thumb. We will later check the details.

**Is this Architecture: -**



**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?
* 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

**Views and Structures in Architecture: -**

* 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: -**

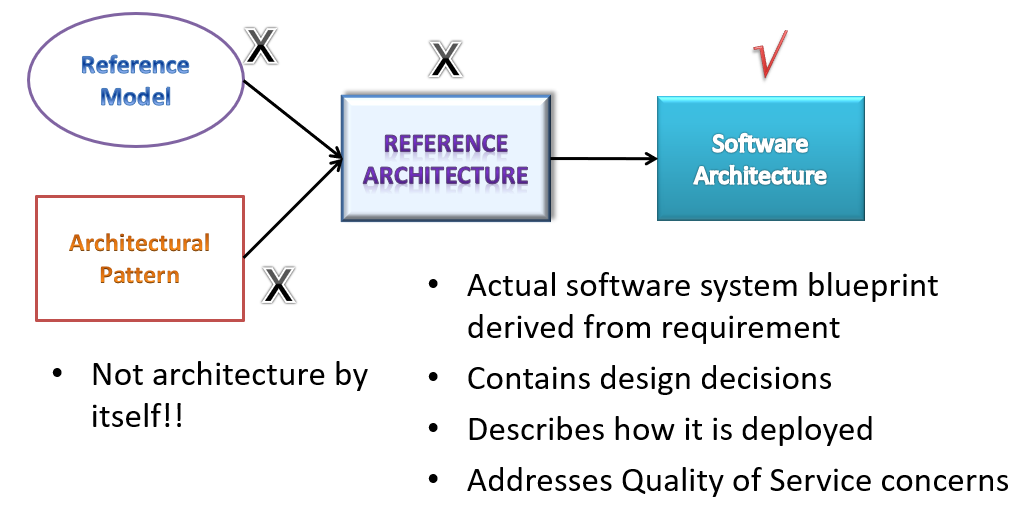
Reference model

* + Decomposes the functionality into a set of smaller units
  + How they interact and share data
  + These units co-operatively implement the total functionality

Reference architecture

* + Derived from the reference model
  + Concrete software elements, mapped to the units of the reference model, that implement the functionality

**Inter-relationships**



**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

**Architecture View: -**

*A representation of a set of elements and the relations among them.*

**Architecture Structure: -**

A structure is set of elements held together by a relation.

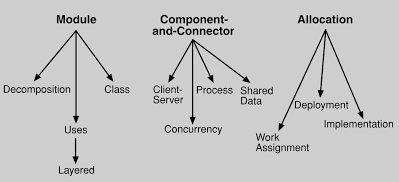
The set of elements itself, as they exist in software or hardware

A software system composes of many structure and always collection of structures represents Architecture.

Three categories of structures –

1. Module
2. Component and Connector
3. Allocation

SOFTWARE STRUCTURES

[](https://sites.google.com/site/softwarearchitectureinpractice/2-what-is-software-architecture/2-5-architectural-structures-and-views/Fig23.png?attredirects=0)

**Module Structure: -**

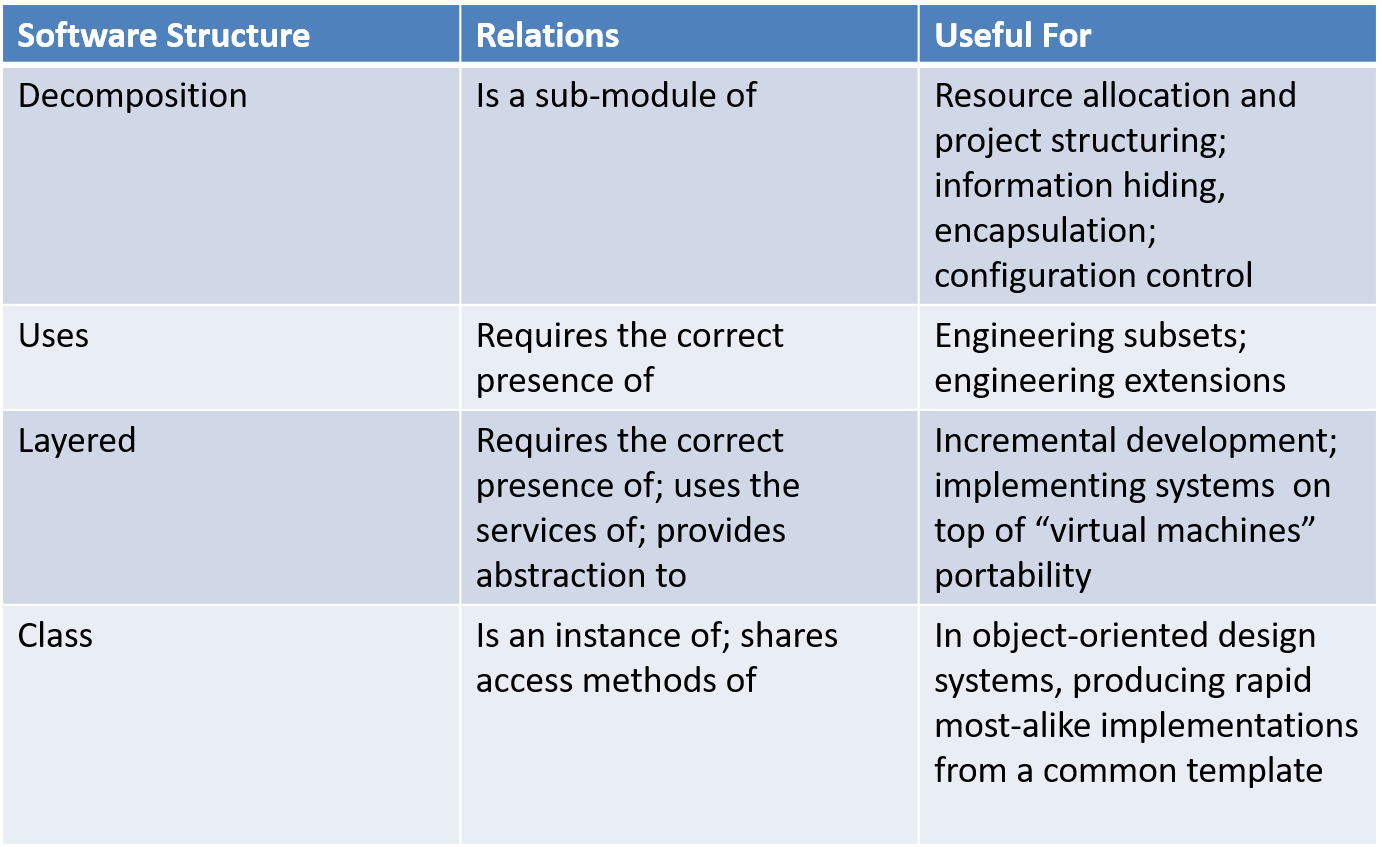
Module system partition systems in to multiple implementation units.

Modules are assigned specific computational responsibilities and based on this work assignment happens for different teams.

In large project – modules are divided in sub modules and assigned to teams.

Elements are modules, which are units of implementation.  
 - What is the primary functional responsibility assigned to each module?  
 - What other software elements is a module allowed to use?  
 - What other software does it use?

* **Decomposition**  
   - shows how larger modules are decomposed into smaller ones recursively
* **Uses**  
    - The units are: modules, procedures or resources on the interfaces of modules  
    - The units are related by the uses relation
* **Layered**  
   - "uses relations" structured into layers
* **Class, or generalization**  
   - shows the “inherits-from” or “is-an-instance-of” relations among the modules
* **Data Model**  
   - The Data model describes the static information structure in the term of data entities and their relationship.



**Component and Connector Structure: -**

This takes care of elements interact with each other at run time. We call this as C&C structure.

Component is always a runtime entity.

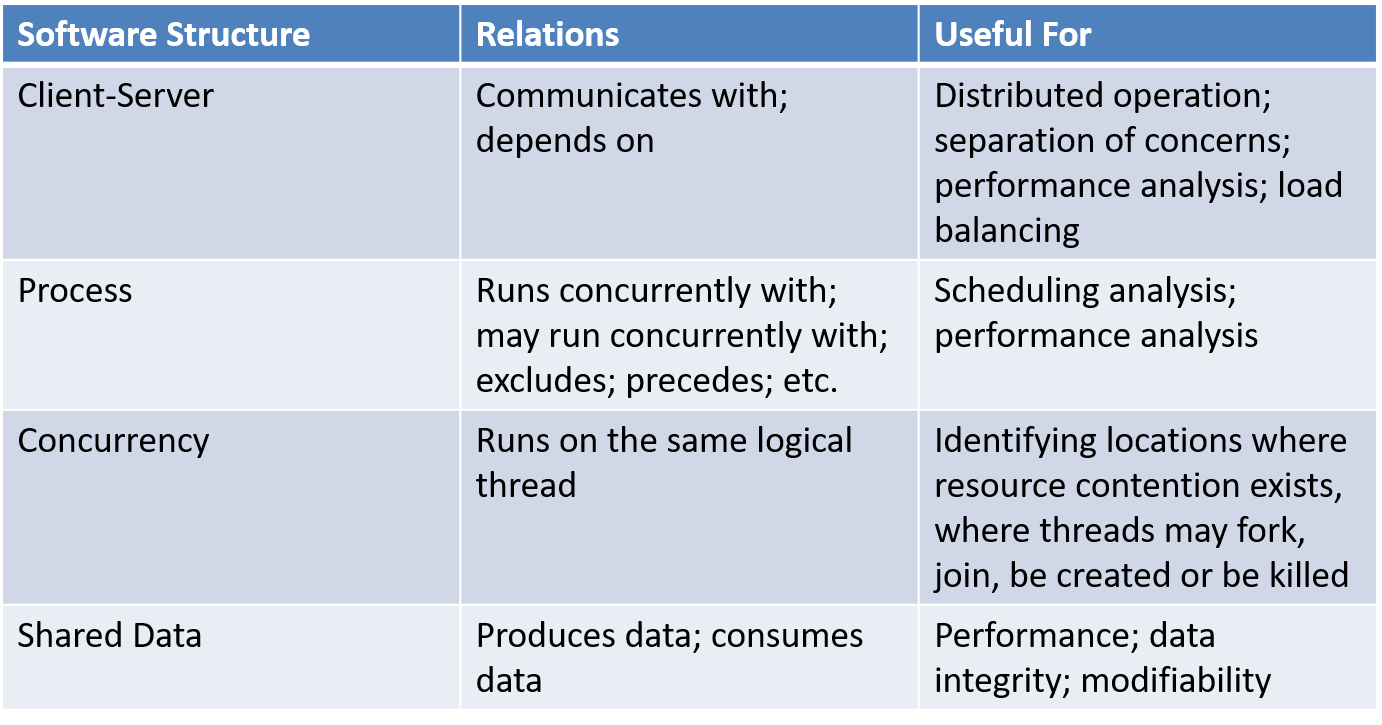
Points –

* System to build set of services. May be more specific to microservices.
* Interaction pattern, synchronization pattern between services describes the system.
* Theses services are made up by various MODULES (implementation units)

Elements are runtime components (units of computation) and connectors (communication vehicles among components)  
The relation is attachment, showing how the components and connectors are hooked together  
 - What are the major executing components and how do they interact?  
 - What are the major shared data stores?   
 - Which parts of the system are replicated?   
 - How does data progress through the system?  
 - What parts of the system can run in parallel?  
 - How can the system’s structure change as it executes?

* **Process, or communicating processes**  
   - Units are processes or threads that are connected with each other by communication, synchronization, and/or exclusion operations
* **Concurrency**  
   - The units are components and the connectors are “logical threads”   
   - A logical thread is a sequence of computation that can be allocated to a separate physical thread
* **Shared data, or repository**  
   - This structure comprises components and connectors that create, store, and access persistent data
* **Client-server**  
   - The components are the clients and servers, and the connectors are protocols and messages
* **Service**  
   - The units are services that interoperate with each other by service coordination mechanism such as SOAP.

- Helps Engineer a system composed of components that may have been develop anonymously and independently each other.



**Allocation Structure: -**

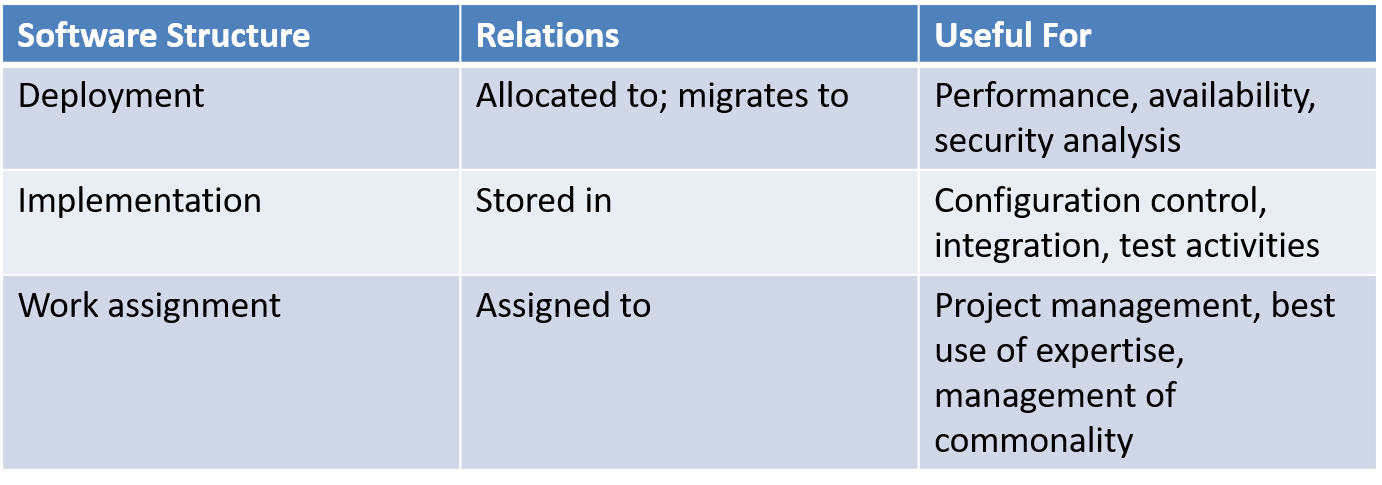
This describe the mapping from software structure to the system environments.

System Environments – Organizational, Developmental, Installation, Execution

Eg. Modules are assigned to teams and assigned to places in a file structure for implementation, integration and testing. Components are deployed on to hardware to execute.

The relationship between the software elements and the elements in one or more external environments  
 - What processor does each software element execute on?  
 - In what files is each element stored during development, testing, and system building?  
 - What is the assignment of software elements to development teams?

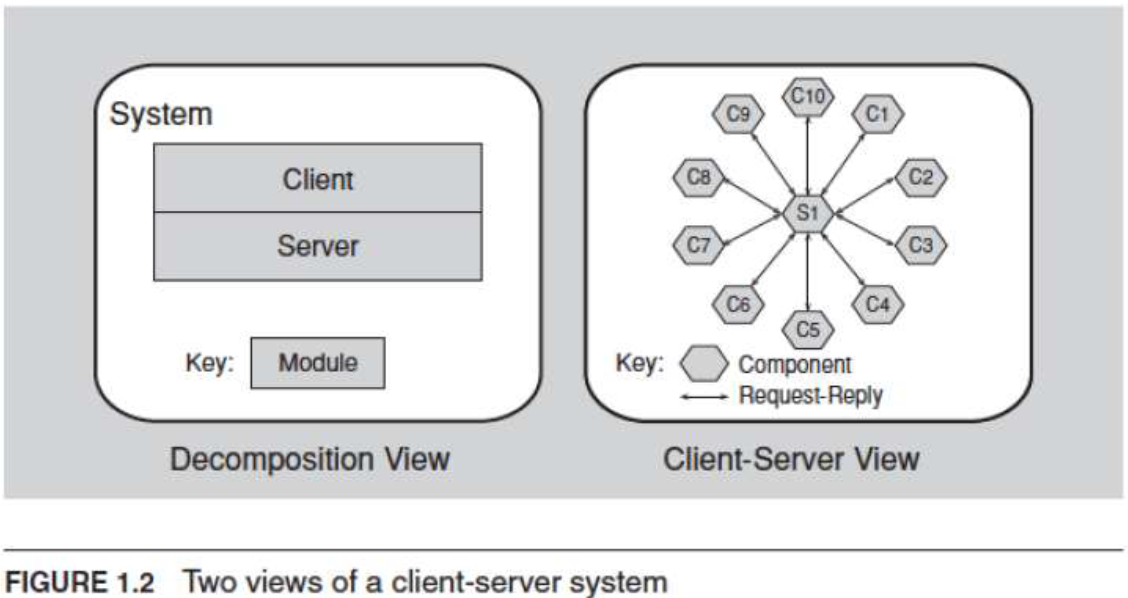
* **Deployment**  
   - Shows how software (usually a process from a component-and-connector view) is assigned to hardware-processing and communication elements  
   - Relations are “allocated-to” and “migrates-to” if the allocation is dynamic
* **Implementation**  
   - How software elements (usually modules) are mapped to the file structure(s)l
* **Work assignment**  
   - Assigns responsibility for implementing and integrating the modules to development teams



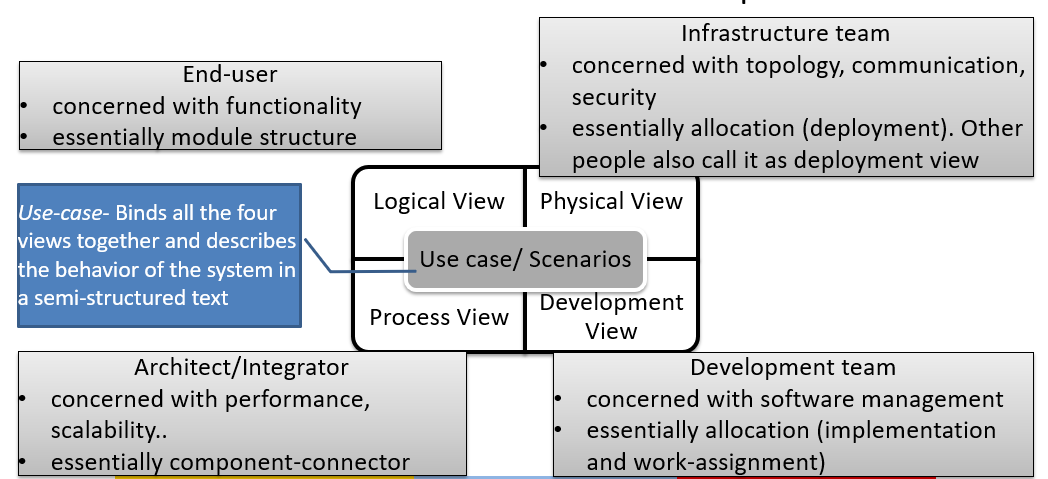
**RELATING STRUCTURES TO EACH OTHER --**

* Elements of one structure are related to elements of other structures
* Often the dominant structure is module decomposition, because it spawns the project structure
* Structures represent a powerful separation-of-concerns approach for creating the architecture
* Structures are basis for architecture documentation

MODULE vs COMPONENT



**4 + 1 view: -**



**Architectural Styles**  
  
Architectural styles tell us, in very broad strokes, how to organize our code. It’s the highest level of granularity and it specifies layers, high-level modules of the application and how those modules and layers interact with each other, the relations between them. Examples of Architectural Styles:  
 Component-based  
 Monolithic application   
 Layered  
 Pipes and filters  
 Event-driven  
 Publish-subscribe  
 Plug-ins  
 Client-server  
 Service-oriented  
  
**Architectural Patterns**  
  
A pattern is a recurring solution to a recurring problem. In the case of Architectural Patterns, they solve the problems related to the Architectural Style. Examples of Architectural Patterns:  
 Three-tier  
 Microkernel  
 Model-View-Controller  
 Model-View-View-Model

**Architectural Pattern: -**

The architectural elements can be composed or work together to solve a particular problem.

* The compositions have been found useful over time, and over many different domains
* They have been documented and disseminated.
* These compositions of architectural elements, called architectural patterns.
* Patterns provide packaged strategies for solving some of the problems facing a system.

***Common Module Type Pattern: --***

Layered Pattern --

* When the uses relation among software elements is strictly unidirectional, a system of layers emerges.
* A layer is a coherent set of related functionality.
* Many variations of this pattern, lessening the structural restriction, occur in practice.

***Common Component and Connector Pattern: -***

* Shared-data (or repository) pattern.
  + This pattern comprises components and connectors that create, store, and access persistent data.
  + The repository usually takes the form of a (commercial) database.
  + The connectors are protocols for managing the data, such as SQL.
* Client-server pattern.
  + The components are the clients and the servers.
  + The connectors are protocols and messages they share among each other to carry out the system’s work.

***Common Allocation Pattern: -***

* Multi-tier pattern
  + Describes how to distribute and allocate the components of a system in distinct subsets of hardware and software, connected by some communication medium.
  + This pattern specializes the generic deployment (software-to-hardware allocation) structure.
* Competence center pattern and platform pattern
  + These patterns specialize a software system’s work assignment structure.
  + In competence center, work is allocated to sites depending on the technical or domain expertise located at a site.
  + In platform, one site is tasked with developing reusable core assets of a software product line, and other sites develop applications that use the core assets.

**Structural “Rules of Thumbs” –**

* The architecture should be the product of a single architect or a small group of architects with an identified technical leader.
  + This approach gives the architecture its conceptual integrity and technical consistency.
  + This recommendation holds for Agile and open source projects as well as “traditional” ones.
  + There should be a strong connection between the architect(s) and the development team.
* The architect (or architecture team) should base the architecture on a prioritized list of well-specified quality attribute requirements.
* The architecture should be documented using views. The views should address the concerns of the most important stakeholders in support of the project timeline.
* The architecture should be evaluated for its ability to deliver the system’s important quality attributes.
  + This should occur early in the life cycle and repeated as appropriate.
* The architecture should lend itself to incremental implementation,
  + Create a “skeletal” system in which the communication paths are exercised but which at first has minimal functionality.
* The architecture should feature well-defined modules whose functional responsibilities are assigned on the principles of information hiding and separation of concerns.
  + The information-hiding modules should encapsulate things likely to change
  + Each module should have a well-defined interface that encapsulates or “hides” the changeable aspects from other software
* Unless your requirements are unprecedented your quality attributes should be achieved using well-known architectural patterns and tactics specific to each attribute.
* The architecture should never depend on a particular version of a commercial product or tool. If it must, it should be structured so that changing to a different version is straightforward and inexpensive.
* Modules that produce data should be separate from modules that consume data.
  + This tends to increase modifiability
  + Changes are frequently confined to either the production or the consumption side of data.

NOTE

* Don’t expect a one-to-one correspondence between modules and components.
* Every process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.
* The architecture should feature a small number of ways for components to interact.
  + The system should do the same things in the same way throughout.
  + This will aid in understandability, reduce development time, increase reliability, and enhance modifiability.
* The architecture should contain a specific (and small) set of resource contention areas, the resolution of which is clearly specified and maintained.

**Summary: -**

* There are three categories of structures:
  + Module structures show how a system is to be structured as a set of code or data units that have to be constructed or procured.
  + Component-and-connector structures show how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors).
  + Allocation structures show how the system will relate to nonsoftware structures in its environment (such as CPUs, file systems, networks, development teams, etc.).
* Structures represent the primary engineering leverage points of an architecture.
* Every system has a software architecture, but this architecture may be documented and disseminated, or it may not be.
* There is no such thing as an inherently good or bad architecture. Architectures are either more or less fit for some purpose.

**Importance of Software Architecture**

**Importance of Architecture: -**

1. An architecture will inhibit or enable a system’s driving quality attributes.
2. The decisions made in an architecture allow you to reason about and manage change as the system evolves.
3. The analysis of an architecture enables early prediction of a system’s qualities.
4. A documented architecture enhances communication among stakeholders.
5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
6. An architecture defines a set of constraints on subsequent implementation.
7. The architecture dictates the structure of an organization, or vice versa.
8. An architecture can provide the basis for evolutionary prototyping.
9. An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule.
10. An architecture can be created as a transferable, reusable model that form the heart of a product line.
11. Architecture-based development focuses attention on the assembly of components, rather than simply on their creation.
12. By restricting design alternatives, architecture channels the creativity of developers, reducing design and system complexity.
13. An architecture can be the foundation for training a new team member.

**Functionality and Architecture: -**

* Functionality and quality attributes are orthogonal.
* It is possible to independently choose a desired level of each quality attribute
* What is functionality?

- It is the ability of the system to do the work for which it was intended.

* Functionality can be achieved with many diffrent architectures, but the software architecture constrains the allocation of functionality to structure

**Architecture/Architect ---- Manages/Responsible**

1. Reasoning About and Managing Change
2. Predicting System Qualities
3. Enhancing Communication Among Stakeholders
4. Earliest Design Decisions
5. Defining Constraints on an Implementation
6. Influencing the Organizational Structure
7. Enabling Evolutionary Prototyping
8. Improving Cost and Schedule Estimates
9. Transferable, Reusable Model
10. Using Independently Developed Components
11. Restricting Design Vocabulary
12. Basis for Training

**Contexts of Software Architecture**

4 contexts for software architecture – ***Technical, Project life cycle, Business, Professional***

1. ***Technical :*** What technical role does the software architecture play in the system or systems of which it’s a part?

Most Important. Industry best practices.

E.g. web-based, object-oriented, service-oriented, mobility-aware, cloud-based, social-networking-friendly

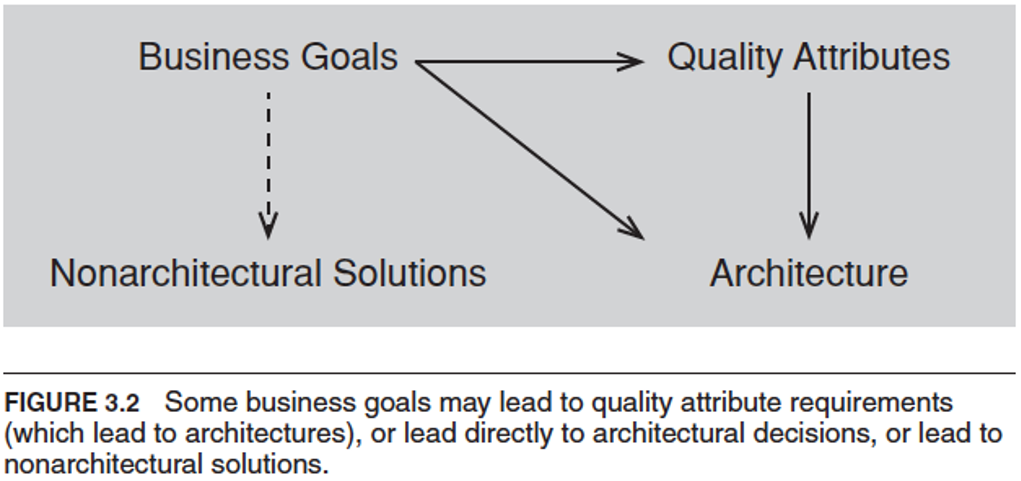
1. ***Project life cycle :*** How does a software architecture relate to the other phases of a software development life cycle?

SDLC Life cycle ---

* + Waterfall
  + Iterative
  + Agile
  + Model-driven development

1. ***Business :***  How does the presence of a software architecture affect an organization’s business environment?

Any thing needed to satisfy business needs. May be speed, response, user friendly.



1. ***Professional :*** What is the role of a software architect in an organization or a development project?

Need skills more than technical skills. Like

* explain to one stakeholder or another the chosen priorities of different properties
* Diplomatic, negotiation, and communication skills
* The ability to communicate ideas clearly
* Manage a diverse workload and be able to switch contexts frequently.
* Up-to-date knowledge on latest technologies.
* Business acumen

**Stakeholders**

A stakeholder is anyone who has a stake in the success of the system.

E.g. – Management, Marketing, End User, Developer, Tester etc.

SH.tiff

**How Architecture is influenced –**

archinf.tiff

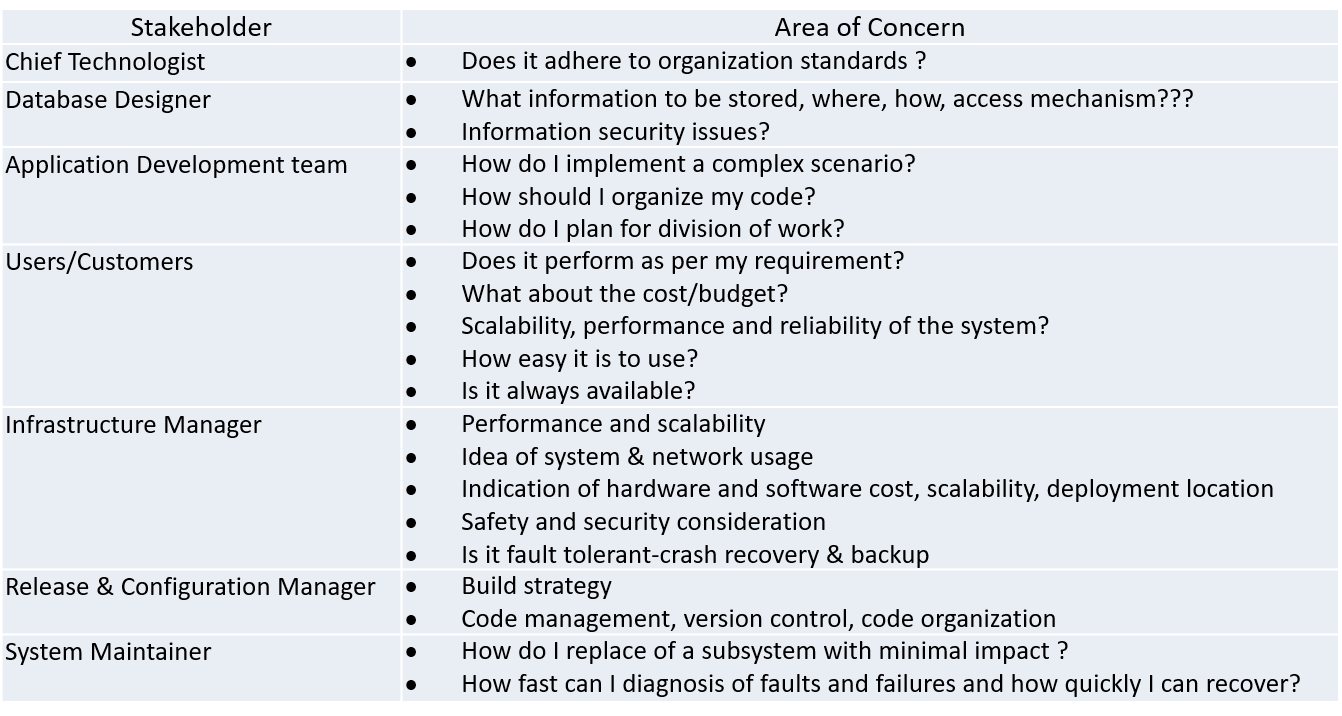
**Architecture Influence Cycle** where Architects **influence back Technical, Project, Business, Professional.**

AIC.tiff

**Summary**

* Architectures exist in four different contexts.
  + Technical. The technical context includes the achievement of quality attribute requirements.
  + Project life cycle. Regardless of the software development methodology you use, you must perform specific activities.
  + Business. The system created from the architecture must satisfy the business goals of a wide variety of stakeholders.
  + Professional. You must have certain skills and knowledge to be an architect, and there are certain duties that you must perform as an architect.
* An architecture has influences that lead to its creation, and its existence has an impact on the architect, the organization, and, potentially, the industry.
* We call this cycle the Architecture Influence Cycle.

**Who are Stakeholders?**



**Architecture and Quality Attributes**

* Quality attributes are achieved through:

architecture AND

implementation / deployment

* Quality attribute requirements significantly influence the architecture
* The degree to which a system meets quality attribute requirements depends on architectural decisions
* The achievement of a quality attribute effects positive or negative the achievement of other quality attributes

security vs reliability

performance vs all other quality attributes

* If architecture decision have nothing to do with quality attribute, you may do somebody else job.
* Important QUALITY attributes are –
  + ***Performance****:* You must manage the time-based behavior of elements, their use of shared resources, and the frequency and volume of inter-element communication.
  + ***Modifiability:*** Assign responsibilities to elements so that the majority of changes to the system will affect a small number of those elements.
  + ***Security:*** Manage and protect inter-element communication and control which elements are allowed to access which information; you may also need to introduce specialized elements (such as an authorization mechanism).
  + ***Scalability:*** Localize the use of resources to facilitate introduction of higher-capacity replacements, and you must avoid hardcoding in resource assumptions or limits.
  + ***Incremental subset delivery:*** Manage inter-component usage.
  + ***Reusability****:* Restrict inter-element coupling, so that when you extract an element, it does not come out with too many attachments to its current environment.

**Three Quality Classes and its attributes: -**

**Specifying Quality Attribute Requirements: -**

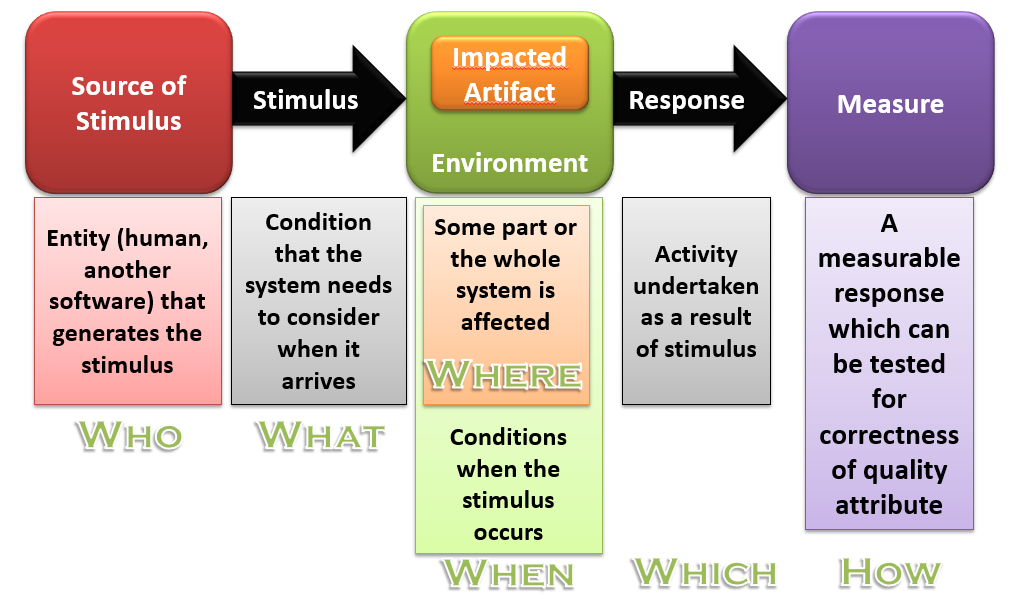
We use a common form to specify all quality attribute requirements as scenarios.

Specifying Parts –

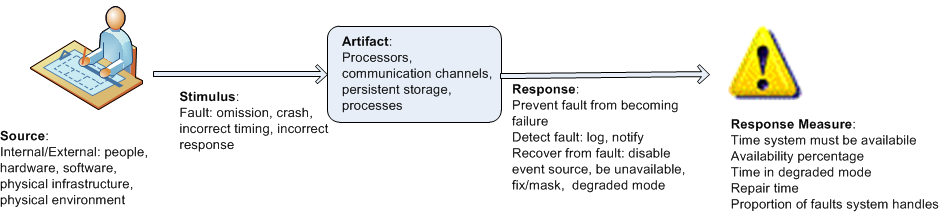
1. **Stimulus**
2. **Stimulus source**
3. **Response**
4. **Response measure**
5. **Environment**
6. **Artifact**

Definitions –

1. **Source of stimulus**. This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
2. **Stimulus**. The stimulus is a condition that requires a response when it arrives at a system.
3. **Environment**. The stimulus occurs under certain conditions. The system may be in an overload condition or in normal operation, or some other relevant state. For many systems, “normal” operation can refer to one of a number of modes.
4. **Artifact**. Some artifact is stimulated. This may be a collection of systems, the whole system, or some piece or pieces of it.
5. **Response**. The response is the activity undertaken as the result of the arrival of the stimulus.
6. **Response measure**. When the response occurs, it should be measurable in some fashion so that the requirement can be tested.



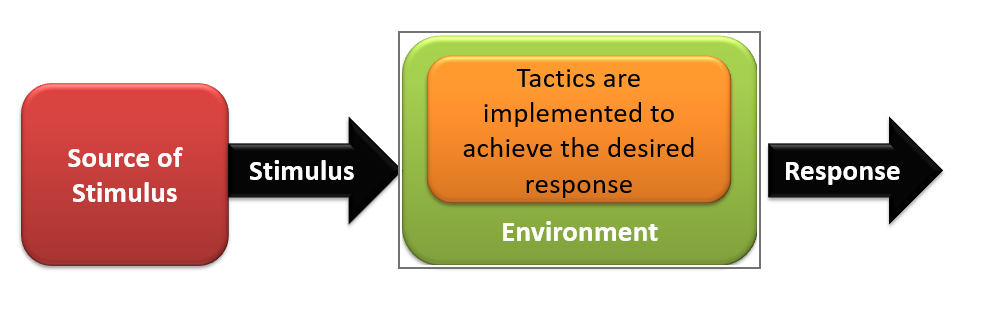
*Example general scenario for availability:*



**Achieving Quality Attribute through Tactics: -**

*TACTICS are collection of primitive design techniques that an architect can use to achieve a quality attribute response.*

* + *Collection of such tactics is* ***architectural strategy***
  + *A pattern can be a collection of tactics*



**Why do we do Tactics? There are three reasons:**

1. Design patterns are complex; they are a bundle of design decisions. But patterns are often difficult to apply as is; architects need to modify and adapt them.
2. If no pattern exists to realize the architect’s design goal, tactics allow the architect to construct a design fragment from “first principles”.
3. By cataloguing tactics, we make design more systematic. You frequently will have a choice of multiple tactics to improve a particular quality attribute. The choice of which tactic to use depends on factors such as tradeoffs among other quality attributes and the cost to implement.

**Guiding Quality Design Decisions: -**

1. Allocation of responsibilities
2. Coordination model
3. Data model
4. Resource Management
5. Mapping among architectural elements
6. Binding time decisions
7. Technology Choice

Note : Easy to remember –

**Allocation of Responsibilities**

* Identifying the important responsibilities including basic system functions, architectural infrastructure, and satisfaction of quality attributes.
* Determining how these responsibilities are allocated to non-runtime and runtime elements (namely, modules, components, and connectors).

**Coordination Model**

* Identify the elements of the system that must coordinate, or are prohibited from coordinating
* Determining the properties of the coordination, such as timeliness, currency, completeness, correctness, and consistency
* Choosing the communication mechanisms that realize those properties. Important properties of the communication mechanisms include stateful vs. stateless, synchronous vs. asynchronous, guaranteed vs. non-guaranteed delivery, and performance-related properties such as throughput and latency

**Data Model**

* Choosing the major data abstractions, their operations, and their properties. This includes determining how the data items are created, initialized, accessed, persisted, manipulated, translated, and destroyed.
* Metadata needed for consistent interpretation of the data
* Organization of the data. This includes determining whether the data is going to be kept in a relational data base, a collection of objects or both

**Management of Resources**

* Identifying the resources that must be managed and determining the limits for each
* Determining which system element(s) manage each resource
* Determining how resources are shared and the arbitration strategies employed when there is contention
* Determining the impact of saturation on different resources.

**Mapping Among Architectural Elements**

* The mapping of modules and runtime elements to each other—that is, the runtime elements that are created from each module; the modules that contain the code for each runtime element
* The assignment of runtime elements to processors
* The assignment of items in the data model to data stores
* The mapping of modules and runtime elements to units of delivery

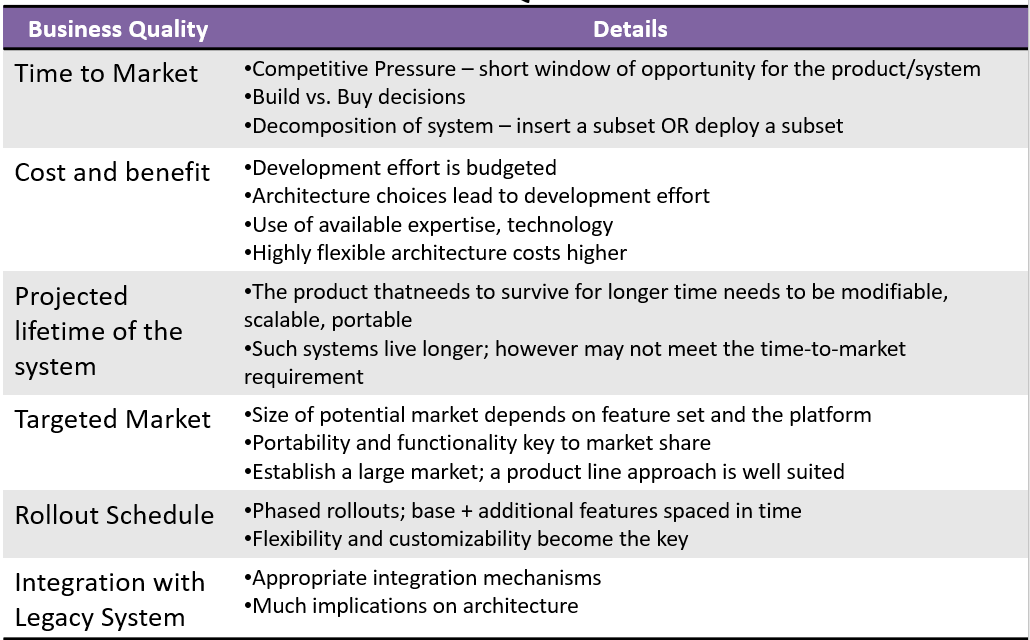
**Binding Time**

* For allocation of responsibilities you can have build-time selection of modules via a parameterized build script.
* For choice of coordination model, you can design run-time negotiation of protocols.
* For resource management you can design a system to accept new peripheral devices plugged in at run-time.
* For choice of technology you can build an app-store for a smart phone that automatically downloads the appropriate version of the app.

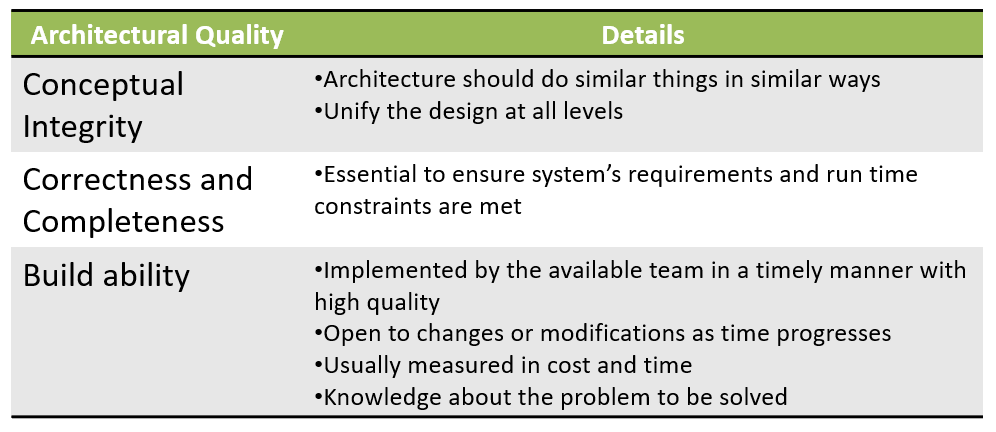
**Choice of Technology**

* Deciding which technologies are available to realize the decisions made in the other categories
* Determining whether the tools to support this technology (IDEs, simulators, testing tools, etc.) are adequate
* Determining the extent of internal familiarity and external support for the technology (such as courses, tutorials, examples, availability of contractors)
* Determining the side effects of choosing a technology such as a required coordination model or constrained resource management opportunities
* Determining whether a new technology is compatible with the existing technology stack

**Business Qualities: -**



**Architectural Qualities: -**



**Availability and Its Tactics**

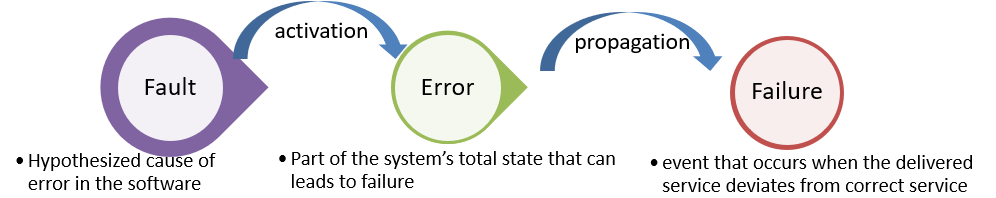
**What is Availability?**

Fundamentally, availability is about minimizing service outage time by mitigating faults. Availability is closely related to security and performance.

Availability is related to reliability.

Example 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



Note: -

Not every fault causes 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 🡪 cost to original system: -

* + Data loss
  + Down-time
  + Cost to fix

**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

**Fault Tree Analysis: -**

It’s an analytical technique that specifies a state of the system that negatively impacts safety and reliability.

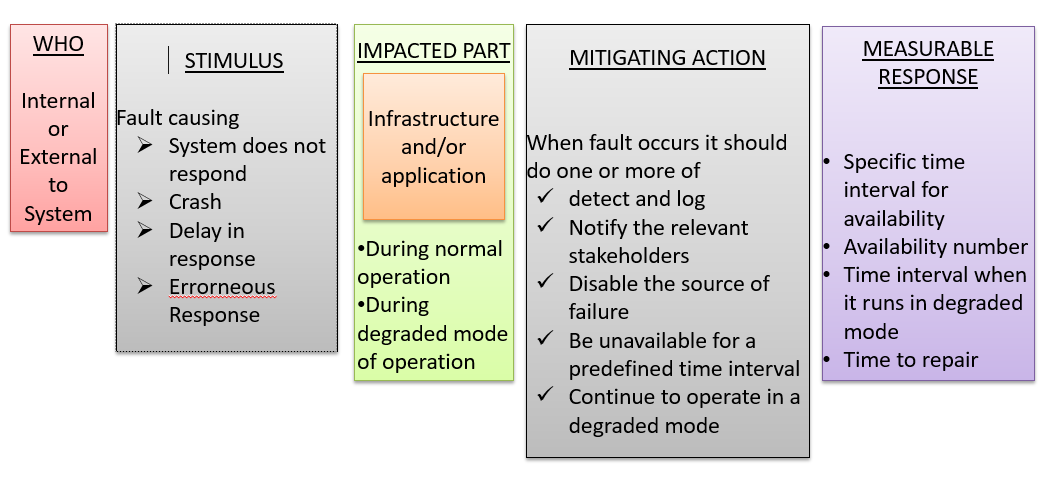
FMECA – (Failure Mode, Effects and Criticality Analysis)

**What is Software Reliability: -**

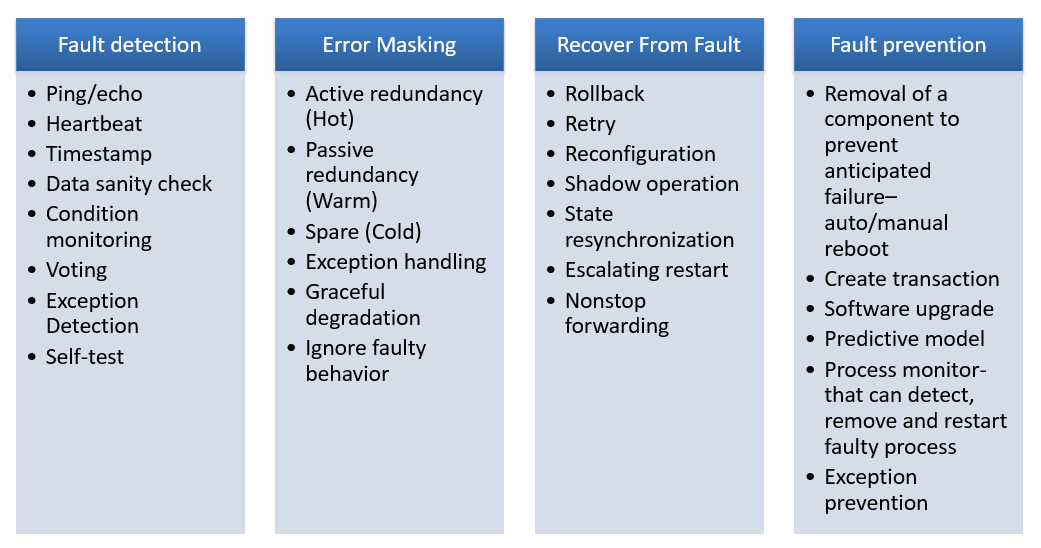
Probability of failure-free operation of a system over a specified time within a specified environment for a specified purpose

* Difficult to measure the purpose,
* Difficult to measure environmental factors.

**Availability Scenarios: -**



**Availability Tactics: -**



**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
* 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 – [Preparation & Repair]**

* Hot spare (Active redundancy)
  + Every redundant process is active
  + When one fails, another one is taken up
  + Downtime is milli-sec
* 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
* 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
* Rollback
* Software upgrade
* Retry
* Ignore Faulty behavior
* Degradation tactics
* Dropping less critical function.
* Reduce functionality rather than making complete system failure.

**Availability Tactics- Fault Recovery –[Reintroduction]**

* 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
* 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 (Removal from service)
  + 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
* Increase competence set
* Designing to handle more cases of fault as a part of normal operations.

**Availability Design Decisions: -**

Allocation of Responsibility

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

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

Mapping Among Architectural Elements

* Determine which artifacts may produce fault.
* Which process need to reassign at runtime
* Mapping modules to redundant components.
* How quickly system can be re installed.
* How data on failed storage can be available by replacement unit.

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

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!!

**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

**Usability and Its Tactics**

**What is Usability: -**

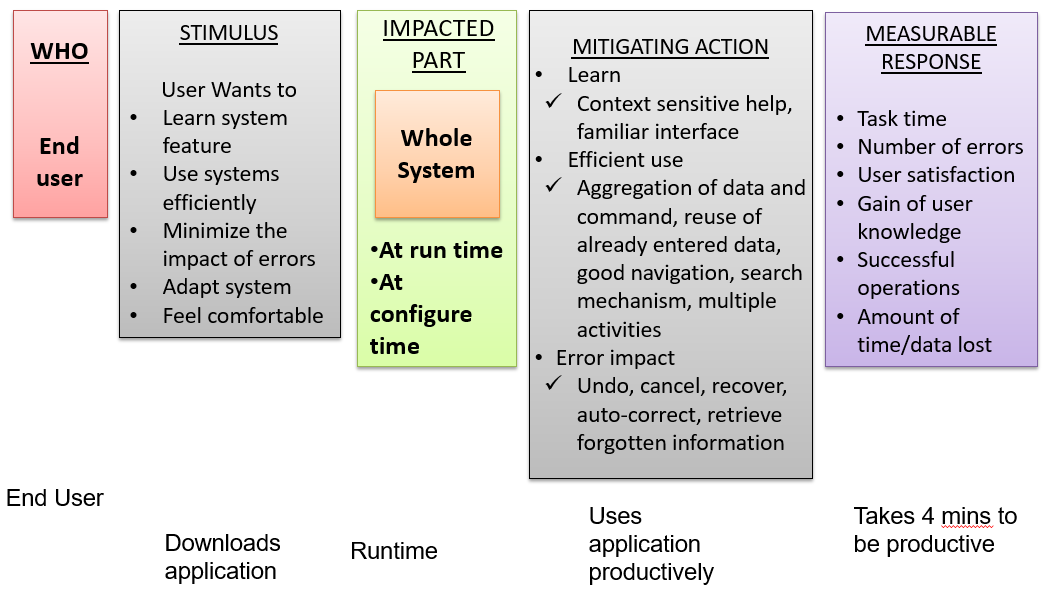
How easy it is for the user to accomplish a desired task and user support the system provides

* 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 comprises following areas –

* Learning system features
* Using a system efficiently
* Minimizing the impact of the error
* Adapting the system to user needs
* Increasing the confidence and satisfaction

**Usability Scenario: -**



**Usability Tactics: -**

**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

**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

**Usability Design Decisions: -**

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
  + Learning how to use the system

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 abstractions needed for undo, cancel
* Design of transaction granularity to support undo and cancel

Mapping Among Architectural Elements

Mapping the architectural elements is visible to the end user.

Resource Management

* Design how user can configure system’s use of resource
* Resource limitations should not make use not achieving the task. For example – attempt to avoid configurations that would result excessively long response time.

Binding Time

* User can choose at run time the system configuration or communication protocol
* Run time selection should not impact user ability to learn.

Technology Choice

* To achieve usability
* Technology that creates online help, produces training materials and collecting feedbacks.

**Modifiability and Its Tactics**

**What is 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

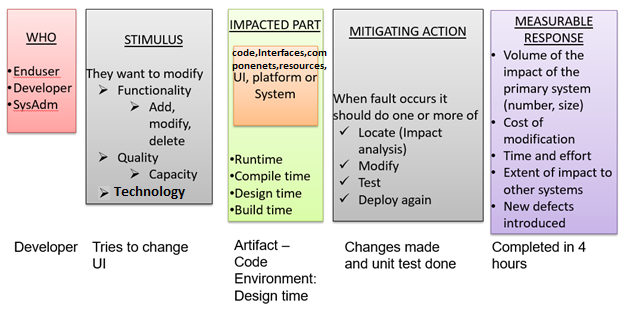
Look for the answer on modifiability: -

* What are changes?
* What is likelihood of the change?
* When is the change made and who makes it?
* What is the cost of the change?

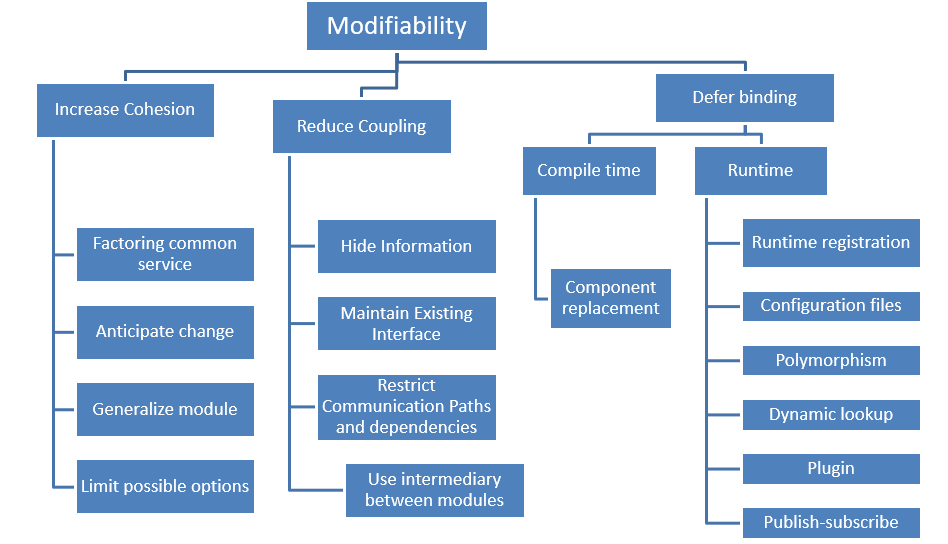
= Cost of introducing mechanism to make the system modifiable

= Cost of making the modifications using the above mechanism

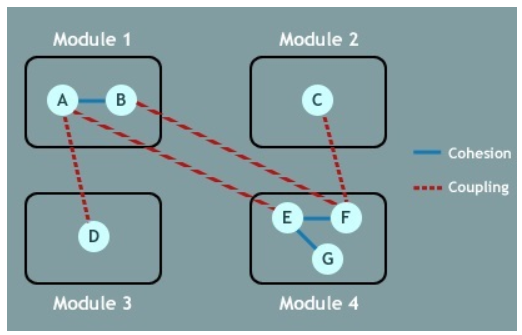
**Modifiability Scenarios: -**

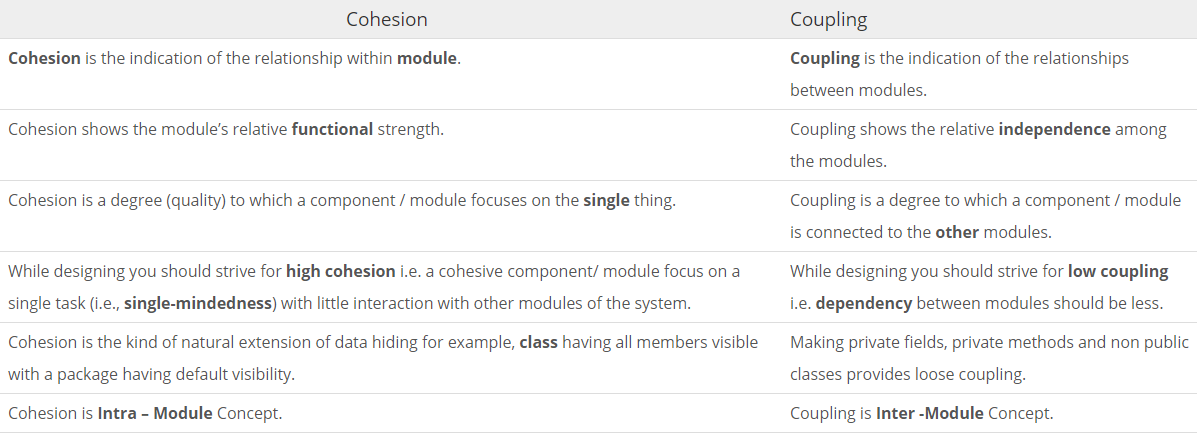


**Modifiability Tactics: -**



**Cohesion vs Coupling: -**





**Reduce Size of a Module Tactics: -**

Split Module: If the module being modified includes a great deal of capability, the modification costs will likely be high. Refining the module into several smaller modules should reduce the average cost of future changes.

**Increase Cohesion Tactics: -**

Increase Semantic Coherence: If the responsibilities A and B in a module do not serve the same purpose, they should be placed in different modules. This may involve creating a new module or it may involve moving a responsibility to an existing module.

**Reduce Coupling: -**

Encapsulate: Encapsulation introduces an explicit interface to a module. This interface includes an API and its associated responsibilities, such as “perform a syntactic transformation on an input parameter to an internal representation.”

Use an Intermediary: Given a dependency between responsibility A and responsibility B (for example, carrying out A first requires carrying out B), the dependency can be broken by using an intermediary.

Restrict Dependencies: Restricts the modules which a given module interacts with or depends on.

Refactor: Undertaken when two modules are affected by the same change because they are (at least partial) duplicates of each other.

Abstract Common Services: Where two modules provide not-quite-the-same but similar services, it may be cost-effective to implement the services just once in a more general (abstract) form.

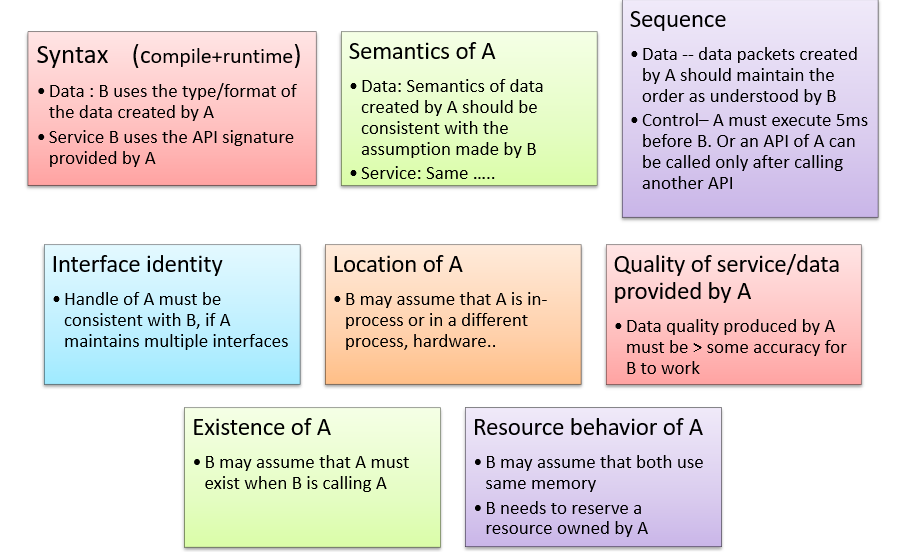
**Defer Binding: -**

In general, the later in the life cycle we can bind values, the better.

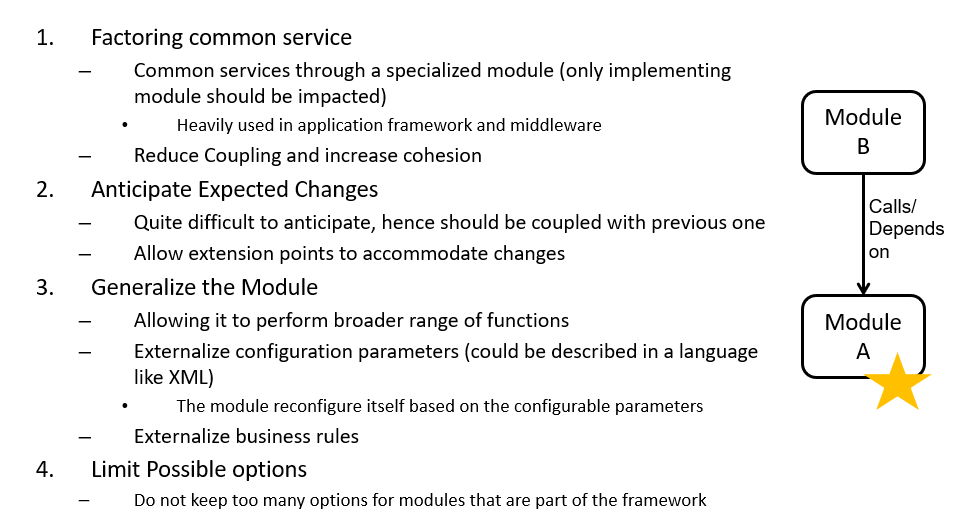
If we design artifacts with built-in flexibility, then exercising that flexibility is usually cheaper than hand-coding a specific change.

However, putting the mechanisms in place to facilitate that late binding tends to be more expensive.

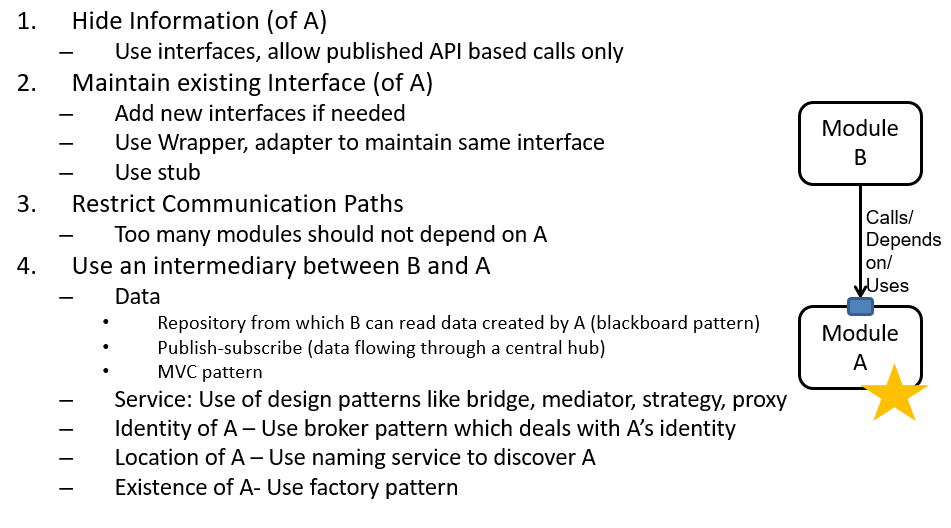
**Dependency between two modules (B🡪 A): -**



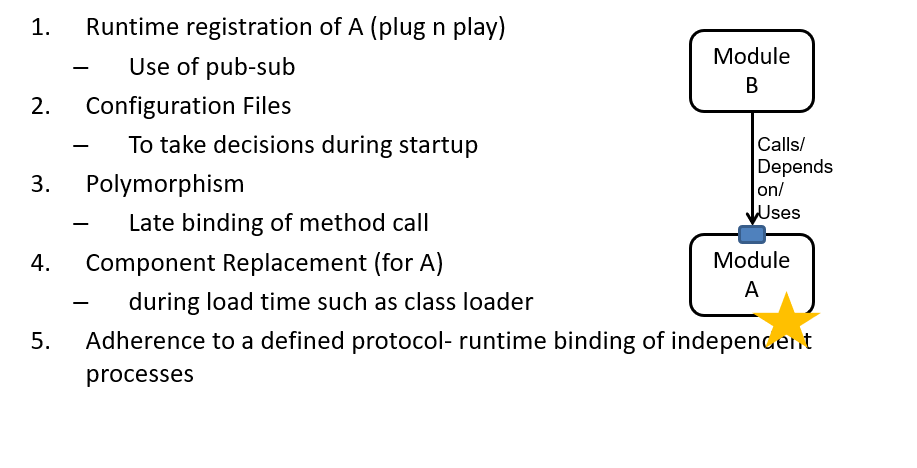
**Localize Modifications: -**



**Prevent Ripple Effect Tactics**



**Defer Binding Time**



**Modifiability – Design Checklist: -**

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

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

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

Technology Choice

* + Evaluate the technology that can handle modifications with least impact (e.g. enterprise service bus)
  + Watch for vendor lock-in

**Performance and Its Tactics**

**What is Performance?**

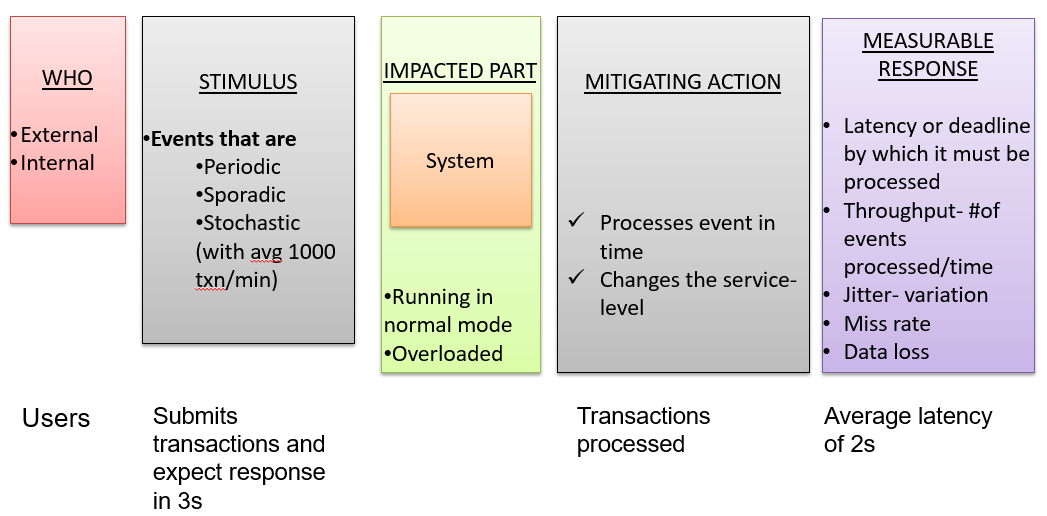
Software system’s ability to meet timing requirements when it responds to an event

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

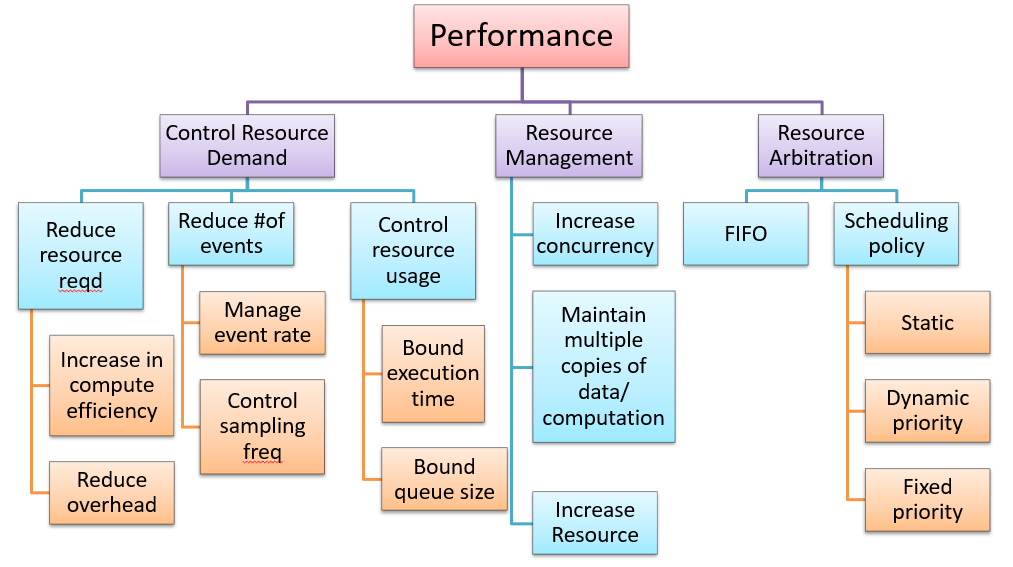
**Performance Scenarios: -**



**Events Categorization: -**

* 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 transaction processed/unit of time
* Jitter- allowable variation in latency
* Number of events not processed not processed because system was too busy to respond

**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 – Scheduling Policies: -**

* 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

**Performance Design Checklist: -**

Allocation of 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

Data Model

* 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.

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

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

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?

**Security and Its Tactics**

**What is Security?**

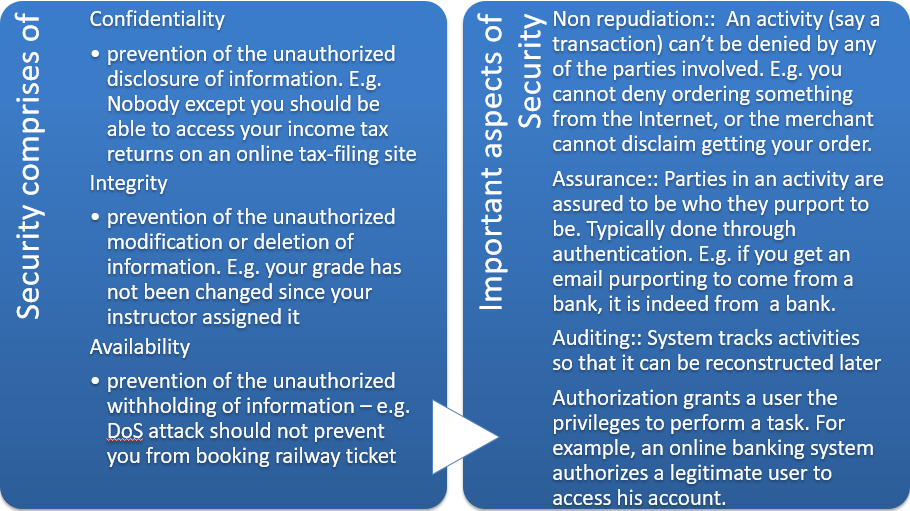
A measure of the system’s ability to resist unauthorized usage while still providing its services to legitimate users.

Aability to protect data and information from unauthorized access

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

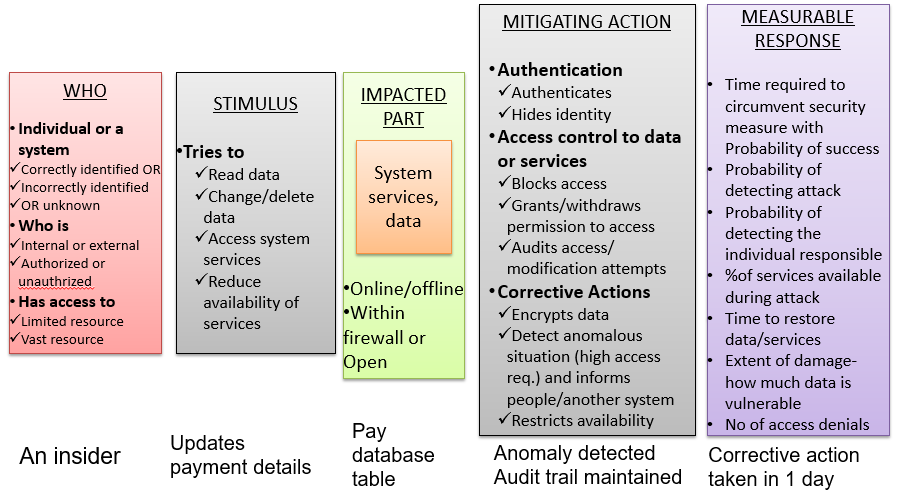
**Important Aspect of Security: -**

Security has three characteristics – CIA – Confidentiality, Integrity and Availability.



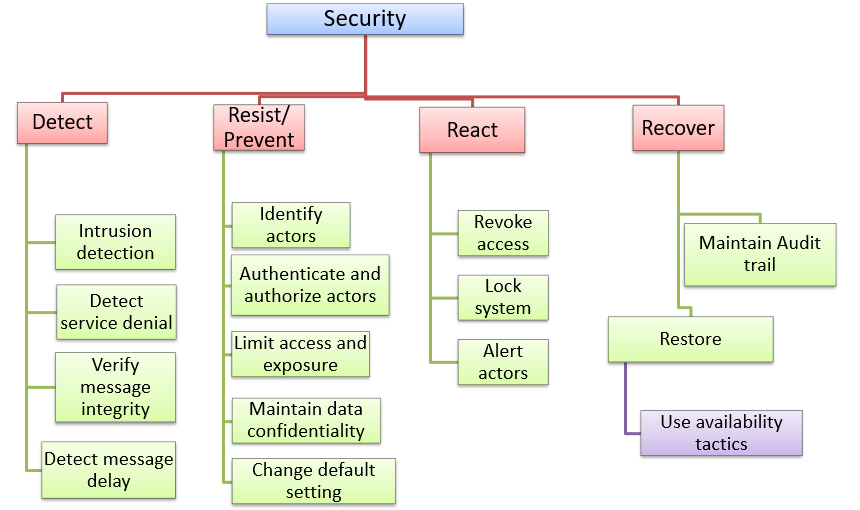
**Security Scenario: -**

The stimulus is an attack.



**Security Tactics**

* 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



**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. **Men in the middle attack**

**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
* 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.

**Security 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

Data Model

* 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.

Coordination Model

* 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.

Resource Management

* 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.

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

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

**Testability and Its Tactics**

**What is Testability?**

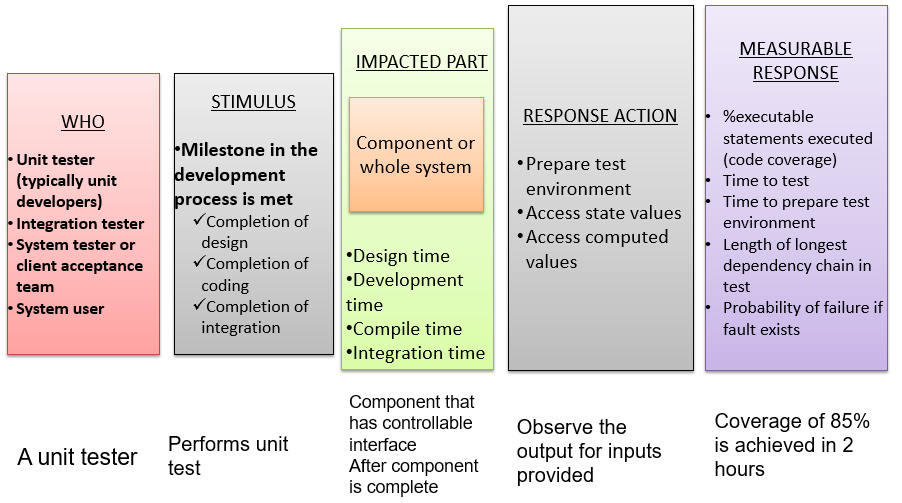
* The ease with which software can be made to demonstrate its faults through testing
* If a fault is present in a system, then we want it to fail during testing as quickly as possible.
* At least 40% effort goes for testing

Done by developers, testers, and verifiers (tools)

* Specialized software for testing
  + Test harness
  + Simple playback capability
  + Specialized testing chamber

**Testability Scenario**

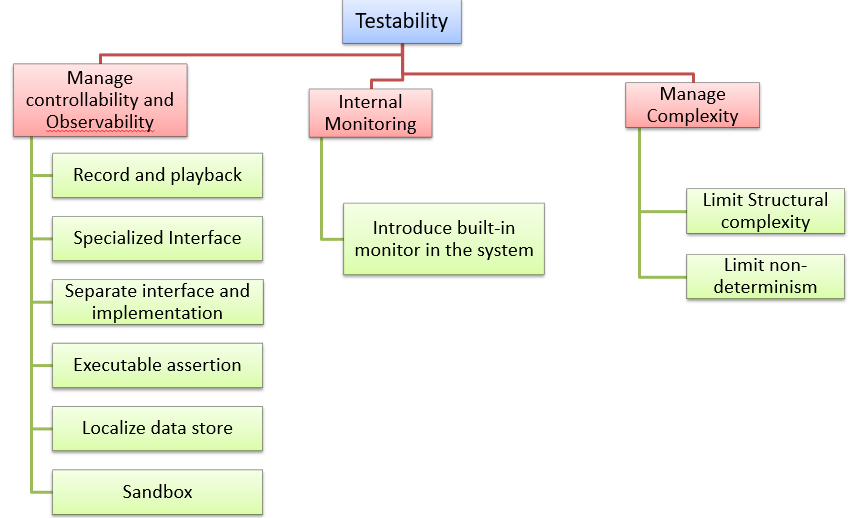
Stimulus inputs are tests.



**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.
* 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

**Testability Design Checklist: -**

Allocation of Responsibilities

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.

Data Model

* 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

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

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.

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

Technology Choice

* 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

**Interoperability and Its Tactics**

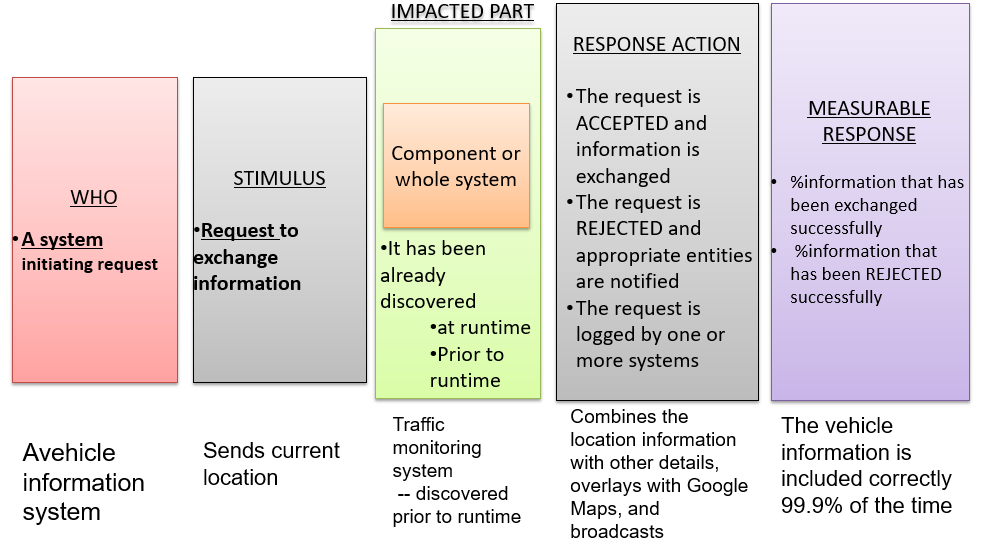
**What is Interoperability: -**

* 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**



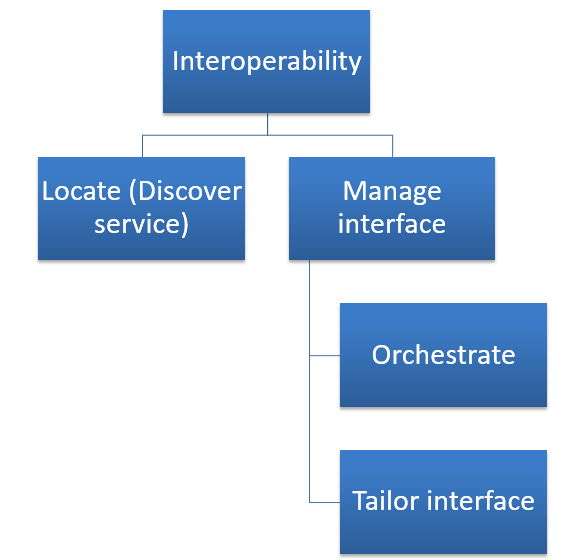
**Notion of Interface**

* Information exchange
  + Can be as simple as A calling B
  + A and B can exchange implicitly w/o direct communication
  + Operation 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

**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
* 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

**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 |

**Interoperability Design Checklist: -**

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

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

Mapping among Architectural Elements

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

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)