🔄 Chapter 5: System Availability

✅ What is System Availability?

* System fails when service is inconsistent with its specification.
* Fault ≠ failure: faults become failures when observable.

🔍 Availability Tactics (Three Types):

1. Fault Detection

* Ping/Echo: ping a component and wait for a response.
* Heartbeat: periodic messages to check status.
* Exceptions: error handling mechanism.

2. Fault Recovery

* Voting: multiple outputs; majority wins.
* Active Redundancy (Hot Restart): all components active; quick switch.
* Passive Redundancy (Warm Restart): backups updated but inactive.
* Spare: standby system takes over with loaded state.
* Checkpoint/Rollback: return to previous safe state.
* Shadow Operation: test recovered component in "shadow mode".

3. Fault Prevention

* Removal from Service: rebooting/preventive action.
* Transactions: undo a series of actions if failure occurs.
* Process Monitor: detect & restart failed processes.

**Chapter 6: System Interoperability – Notes for Long Questions**

**✅ 1. What is Interoperability?**

**Definition**:  
Interoperability means how well **two or more systems** can **exchange useful and meaningful information** using interfaces, within a specific context.

🔹 It's not just about data transfer — it's about **understanding** the information:

* **Syntactically** (structure is correct)
* **Semantically** (meaning is understood)

🔹 Like other quality attributes (e.g., performance, modifiability), **interoperability is not binary** (yes or no). It can vary in **degree or extent**.

**✅ 2. Interoperability – General Scenario**

This is a general template used to define **any quality attribute**, adapted here for interoperability.

| **Element** | **Explanation** |
| --- | --- |
| **Source of stimulus** | The system that **initiates a request** (e.g., client app). |
| **Stimulus** | A **request to exchange information** between systems. |
| **Artifacts** | The systems that want to work together (interoperate). |
| **Environment** | Systems can be discovered **at runtime** or are already known before. |
| **Response** | Request leads to exchange of info which is either: |
|  | ✔ Correctly understood (syntax + meaning), OR |
|  | ❌ Rejected + entities notified. Logged either way. |
| **Response Measure** | Measured by: |
|  | - % of info correctly processed |
|  | - % of info correctly rejected |

**✅ 3. Interoperability Tactics (How to Achieve It)**

Tactics are design techniques to achieve a quality attribute (here: interoperability). These are divided into two categories:

**🔹 A. Locate**

**Tactic: Discover Service**

🔸 Used when systems must find each other **at runtime**.

🔸 This means the systems are **not hardcoded** in advance.

🔸 Example:  
A mobile app finds nearby Bluetooth or WiFi devices to connect to.

🔸 **How?**  
Locate the service using a **directory service** (like UDDI for web services).

**🔹 B. Manage Interfaces**

2 tactics fall under this:

**1. Orchestrate**

* A control mechanism is used to **coordinate and manage** the interaction between systems.
* It controls the **order** in which services are called, and how they respond.
* **Use this when** systems have to perform **complex tasks together**.
* **Example**:  
  Online travel booking site that coordinates flight booking, hotel booking, and payment gateways.

**2. Tailor Interface**

* Change the interface to **add or remove capabilities**:
  + Add: Translation of data, buffering, smoothing.
  + Remove: Hide sensitive functions from untrusted users.
* **Purpose**: Make interface suitable to match the **other system’s capabilities or user needs**.
* **Example**:
  + API hides admin functions from regular users.
  + Data translated from XML to JSON between two systems.

**✅ 4. Why Standards Are Not Enough for Interoperability**

Although standards exist to make systems work together, in reality **standards alone cannot guarantee interoperability**.

Here’s why:

**🔹 A. Implementation Differences**

* Ideally, all systems that follow the same standard should work together.
* But in practice, standards have **open-ended parts** that can be **customized**.
* This leads to **proprietary implementations** that are not compatible.

**🔹 B. Standards Evolve Over Time**

* Standards get updated — some changes are **backward-compatible**, others are not.
* Organizations must decide **when to upgrade** and **which version** to adopt.
* This creates **mismatches** between different systems.

**🔹 C. Bad or Conflicting Standards**

* Not all standards are good. Many are:
  + **Underspecified**: Missing important details.
  + **Overspecified**: Too strict, limiting flexibility.
  + **Inconsistently specified**: Rules contradict each other.
  + **Unstable** or **irrelevant**: Frequently changing or outdated.
* Sometimes, competing organizations promote **different standards** that conflict.

**🔹 D. Standardization Can Limit Innovation**

* In **new or fast-changing fields**, early standardization can be harmful.
* It can **lock in inferior approaches**, and **prevent better solutions** from being adopted later.

**Chapter 7: System Modifiability – Long Question Notes**

**✅ 1. What is Modifiability?**

**Modifiability** is how **easily and cost-effectively** a software system can **accommodate changes**.

🔹 It reduces:

* **Effort**, **time**, and **cost** needed to apply changes.

**🔍 Modifiability = Cost of Change**

It depends on:

* **What** is changing?
* **When** the change is made?
* **Who** is making the change?

**A. What Can Change?**

Almost anything in a system:

* **Functions** (what it does)
* **Platform** (OS, hardware, middleware)
* **Environment** (systems it connects to, protocols)
* **Qualities** (performance, reliability)
* **Capacity** (users, transactions)

**B. When is the Change Made?**

Changes can be made at:

1. **Source Code** level (Compile time)
2. **Build time** (Choosing libraries)
3. **Configuration time** (Parameter settings)
4. **Execution time** (Runtime changes)

**C. Who Makes the Change?**

* **Developer**: During development
* **End user**: Through UI or settings
* **System administrator**: Through system config or deployment

**✅ 2. Modifiability Tactics**

Tactics are organized into 3 **goal-based** categories:

**🔹 A. Localize Modifications**

**Goal**: Reduce the number of modules affected by a change.

**1. Maintain Semantic Coherence**

* Keep related responsibilities **together in the same module**.
* Use **common services** and **abstractions** to reduce repetition.
* Example: Using a shared logging module instead of rewriting logs in every class.

**2. Anticipate Expected Changes**

* Design your system **while thinking of likely future changes**.
* Helps assign responsibilities smartly.
* Limitation: Not all changes can be predicted, so it works best with semantic coherence.

**3. Generalize the Module**

* Make a module **handle multiple similar tasks**, rather than just one.
* This allows changes by **adjusting inputs**, not by rewriting code.
* Example: A payment module that handles multiple payment methods.

**4. Limit Possible Options**

* Reduce variation in what can change to control complexity.
* Example: Only allow specific types of hardware, OS, or processors.

**🔹 B. Prevent Ripple Effect**

**Goal**: Stop a change in one part from **spreading** to others.

**1. Hide Information (Encapsulation)**

* Divide the system into small parts, **exposing only needed interfaces**.
* Keeps internal changes from affecting other modules.

**2. Maintain Existing Interface**

* Keep the **method name and parameters** the same even if implementation changes.
* Can use:
  + **Adapter**: Wrap the new version with the old interface.
  + **Stub**: A placeholder when removing a module.

**3. Restrict Communication Paths**

* Minimize how many other modules **send/receive data** from this module.
* Reduces the chance of dependency, thus less ripple.

**4. Use an Intermediary**

* Place a **middle layer** between two modules.
* This layer handles communication, translation, or interaction.
* Patterns used:
  + **Facade**, **Proxy**, **Mediator**, **Strategy**, **Factory**
  + **Name Server**: Helps locate a service at runtime without hardcoding location

**🔹 C. Defer Binding Time**

**Goal**: Delay applying changes so **end users or admins** can do it easily.

**Key Concepts**

* Helps with:
  + **Faster deployment**
  + **Changes without rebuilding**
  + **Giving control to non-developers**

**Techniques**

* **Runtime registration**: Plug-and-play style.
* **Publish/Subscribe**: Dynamically attach services or data handlers.
* **Configuration files**: Set parameters at startup.
* **Polymorphism**: Decide method behavior at runtime.
* **Component replacement**: Replace parts of system without modifying code.
* **Standard protocols**: Allow independent services to work together at runtime.

**✅ 3. Modifiability Case Study – Subway System**

Scenario:

* Ticket machine accepts **cash** but gives **no change**.
* Separate machine gives **change**, but doesn't sell tickets.
* 6–8 ticket machines for every change machine.

**Tactics at Play:**

* **Limit options**: Users have limited, fixed behaviors — reduces complexity.
* **Hide Information**: Machines do not share internals, they just perform specific jobs.
* **Defer Binding Time**: User chooses when to get change.

**Chapter 8: System Performance – Long Question Notes**

**✅ 1. What is System Performance?**

**Performance** is about how quickly a system **responds to events**.

An event can be:

* A user request
* Message arrival
* Timer expiration
* Environmental change

**The system should process events and give a response within some time constraint.**

**Two Main Contributors to Response Time**

| **Type** | **Explanation** |
| --- | --- |
| **Resource Consumption** | Time spent actively computing / processing |
| **Blocked Time** | Time spent waiting due to unavailability, contention, or dependency |

**✅ 2. Tactics to Improve Performance**

Divided into 3 Categories:

**🔹 A. Resource Demand**

**Goal**: Reduce how much of a resource is needed.

**1. Increase Computational Efficiency**

* Use **better algorithms** to reduce processing time.
* Example: Use optimized queries in databases.
* Trade resources: Regenerate data or store intermediate results depending on memory/time.

**2. Reduce Computational Overhead**

* Avoid unnecessary processing.
* Remove middle layers (used for modifiability) that add delay.
* Example: Removing an extra API gateway if it’s not required.

**🔹 B. Resource Management**

**Goal**: Better **use of existing resources**.

**1. Introduce Concurrency**

* Use **threads** or **parallel processing**.
* Process multiple requests at the same time.

**2. Maintain Multiple Copies**

* **Replicate data or computations** to reduce wait time.
* Examples:
  + Replicated servers
  + Caching: Store frequently used data locally

**3. Increase Available Resources**

* Add more:
  + CPUs
  + Memory
  + Bandwidth
* **Costly**, but reduces latency effectively.

**🔹 C. Resource Arbitration**

**Goal**: Schedule how resources are given to different tasks.

**Key Concepts**

* Every resource (CPU, buffer, network) is **shared**, so it needs **scheduling**.
* A good schedule = fast response + fairness.

**1. First-in First-out (FIFO)**

* All requests treated equally, **served in order**.
* Problem: A fast request might get stuck behind a long one.

**2. Fixed Priority**

* Each request has a **static priority**.
* High-priority tasks are served first.
* Problem: **Starvation** of low-priority tasks.

**3. Dynamic Priority**

* **Round Robin**: Every task gets a time slice, then moves to the back.
* **Earliest Deadline First**: The task with the closest deadline is served first.

**✅ 3. Performance Case Study – Retail Store**

**Goal**: Speed up checkout in a physical store.

**Resource Management Examples:**

* **Introduce concurrency**: More counters → parallel processing.
* **Replicate data/computation**: Put common items at multiple places.
* **Increase resources**: Add more cash registers and barcode scanners.