Availability

Readiness to perform tasks when needed, extends reliability by including recovery, minimize service outages.

Availability General Scenario

Portion of Scenario	Possible Values
Source	Internal/external: people, hardware, software, physical infrastructure, physical environment
Stimulus	Fault: omission, crash, incorrect timing, incorrect response
Artifact	Processors, communication channels, storage, processes, affected artifacts
Environment	Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
Response	Prevent the fault from becoming a failure Detect the fault: • log the fault • notify appropriate entities (people or systems) Recover from the fault • disable source of events causing the fault • be temporarily unavailable while repair is being effected • fix or mask the fault/failure or contain the damage it causes • operate in a degraded mode while repair is being effected
Response Measure	Time or time interval when the system must be available Availability percentage (e.g. 99.999%) Time to detect the fault Time to repair the fault Time or time interval in which system can be in degraded mode Proportion (e.g., 99%) or rate (e.g., up to 100 per second) of a certain class of faults that the system prevents, or handles without failing

EXERCISE/EXAMPLE

1. Source of stimulus: Server and database /Hardisk
2. Stimulus: Server doesn't deliver /Crashes
3. Environment: /Normal operation
4. Artefact: /Database
5. Response: /System notifies operator, redundant server takes over
6. Response measure: /No downtime

Availability Tactics

Help systems withstand faults, ensure services meet specifications by preventing, limiting or repairing fault impacts. Based on different scenarios, ensure that is mostly available.

- Detect Faults
    - Monitor that tracks the health of the system, detecting failures or congestion
    - Ping/Echo: message exchange to test network reachability and measure round-trip delay

    - Heartbeat: periodic message send to monitor the status of a system.(one-way)
    - Timestamp: identify incorrect event sequences
    - Condition Monitoring: monitor state to ensure proper functioning (it knows what the output is supposed to be or something like that, predictive maintenance and failure prevention)
    - Sanity Checking: Verifying that the system is functioning between the expected parameters or boundaries (error detection during execution)
    - Voting: ensures replicated components produce the same result
    - Exception Detection: identify anomalies or failures that deviate from expected system behavior
    - Self-test: a component verifies its own functionality (using things like checksums)
- Recover from Faults
    - Redundant spare: a duplicate component can take over if primary component fails
    - Rollback: revert to a previous safe state
    - Exception Handling: manage errors of unexpected events during execution
    - Software Upgrade: runtime upgrades. Functional path (store updated software to pre-allocated RAM), class patch (add data and methods to classes at runtime), hitless in-service upgrade (use redundant spare for seamless in-service updates).
    - Retry: attempt a failed operation again
    - Ignore Faulty Behavior: disregard minor errors to maintain system operation
    - Graceful Degradation: continue functioning at reduced capacity despite failure
    - Reconfiguration: adjust system components to maintain functionality during failures or changes
    - Shadow: Run a component in the background and try to fix it before reactivating it
    - State Resynchronization: (works with rollback) synchronize state of the active and standby components
    - Escalating Restart: progressively restart components to minimize service disruption
    - Non-stop forwarding: uninterrupted data flow by maintaining alternative paths during failures.

- Prevent Faults
    - Removal From Service: temporarily take a component offline to prevent it from affecting the rest of the system
    - Transactions: group operations into a single atomic unit (fully completes or fully fails)
    - Predictive Model: monitor process health and use previous history to avoid faults
    - Exception Prevention: mask faults or use smart pointers and wrappers to avoid exceptions
    - Increase Competence Set: expand capabilities to handle a broader range of tasks or challenges

Patterns for Availability

- Active Redundancy (hot spare): nodes processes identical in parallel, keeping redundant spares synchronized, when failure occurs another can take over instantly. For critical systems like air traffic control or financial transactions.
- Passive redundancy (warm spare): only the main (active) node processes input, while spares are updated with state information periodically. Backup is partially ready. For cloud-based applications.
- Spare (cold spare): redundant spares stay inactive until failover, slow recovery. For const -sensitive applications with low availability demands.
- Triple Modular Redundancy (TMR): three identical components running in parallel that use majority voting to detect and mask errors (output with 2/3 votes is trusted). For aerospace and safety-critical systems.
- Circuit breaker: keep a service from trying countless retries, giving it time to recover, maintains system responsiveness under partial failure. For microservices and distributed systems, payment gateways.
- Process Pairs: a primary process works normally, while a backup process keeps checkpoint data, so if primary fails the backup rolls back to the last good state. For databases and telecommunication systems.
- Forward Error Recovery: move from undesirable state to desirable one with error correction mechanisms to correct and continue without reverting, fixing errors on the fly. For real-time systems, error-tolerant computing.

Common combinations

Combination	**Why it Works**
Circuit Breaker + Passive Redundancy	Microservices redirect to warm spare instance during downtime and avoid retry storms

Combination	Why it Works
Process Pairs + Forward Error Recovery	Backup process can correct partial state corruption and resume from last checkpoint
TMR + Forward Error Recovery	When even majority voting reveals anomalies, error correction kicks in for soft faults (e.g., radiation)
Active Redundancy + Circuit Breaker	If one active path is failing, circuit breaker isolates it while another active node continues processing

Conclusion guide

Use Active Redundancy when you need **instant failover with no downtime**, especially for **mission-critical systems** like air traffic control or trading platforms.

Use Passive Redundancy (Warm Spare) for systems needing **fast recovery** but where **cost or complexity of full-time redundancy** is a concern (e.g., enterprise cloud systems).

Use Cold Spare for **cost-sensitive systems** where availability is **not critical** and **recovery time is acceptable** (e.g., archival services, internal tools).

Use Triple Modular Redundancy (TMR) for **safety-critical systems** that must continue even if one component fails silently, such as in **aerospace or nuclear control**.

Use Circuit Breaker in **distributed or microservice architectures** to prevent cascading failures and maintain responsiveness during partial outages.

Use Process Pairs for systems requiring **stateful recovery**, where it's critical to return to the **last known good state**, like in databases or telecom.

Use Forward Error Recovery when the system must **continue operating through faults**, correcting them dynamically (e.g., embedded real-time systems, autonomous vehicles).

Deployability

Ability to make a system available for a user within a prodictable time, budget and effort. Architect needs to consider how it is measured, executed, updated, monitored and controlled.

How does architecture support deployment: Granularity (deploy entire system or individual elements), Control (different granularity levels, monitoring and rollback) and Efficiency (rapid deployment with minimal effort).

Continuous Deployment: fully automated deployment spans from coding to user interaction with the product.

Deployment environments: Development, Integration (build executable services), Staging (testing system qualities), Production (close monitoring)

Deployment General Scenario

Portion of Scenario	Possible Values
Source	End user, developer, system adm, operations personnel, component marketplace, product owner
Stimulus	New element available to be deployed: Typically, a request to replace software element with new version (fix defect, security patch, upgrade of component/framework, upgrade internal elements) New element approved for incorporation. Existing element(s) needs to be rolled back.
Artifact	Specific components or modules, system's platform, UI, its environment, another system it interoperates with. Artifact might be single or multiple software elements, or entire system.
Environment	Full deployment, Subset to specified portion of users, VMs, Containers, Servers, Platforms
Response	Incorporate new components. Deploy new components. Monitor new components. Roll back previous deployment.
Response Measure	Cost in terms of: • Number, size, and complexity of affected artifacts • Average/worst-case effort • Elapsed clock or calendar time • Money (direct outlay or opportunity cost) • New defects introduced Extent to which this deployment/rollback affects other functions or quality attributes. Number of failed deployments. Repeatability/Traceability/Cycle time of the process.

EXERCISE/EXAMPLE

Source: End user, code repository / End user

Stimulous: book two tickets for different movies at the same time /Request for security update

Artifact: /Payment component

Environment: /Production

Response: /New version of component tested

Response Measure: /Withing 1 day and 6 person-hours effort

Deployability Tactics

- Manage Deployment Pipeline

    - Scale Rollouts: gradually deploy a new service version to a controlled subset of users (alpha, beta, release)
    - Roll back: revert to a previous version if the deployment has defects or fails to meet expectations
    - Script deployment commands: use scripts documented, reviewed, tested and controlled for deployment (with scripting engine automatic execution)
- Manage Deployed System
    - Manage Service interactions: enable simultaneous deployment and execution of versions (coexistence of functioning versions)
    - Package dependencies: bundle elements with correct versions of dependencies for seamless deployment from development to production
    - Feature toggle: include a runtime "kill switch" to disable new features automatically if needed

Deployability Patterns

- Microservice Architecture: organize the system as a collection of independent services that communicate through message-based interfaces. Each service can be deployed and scaled independently. For large, fast-evolving systems like e-commerce platforms and media services.
- Replacement of services:
    - Blue/Green deployment: two environments run in parallel – blue (current version) and green(new version). Traffic is switched to green once it's verified. When new instance work satisfactory remove old version (blue); if there is a problem roll back to blue. For bank APIs, Ci/CD pipelines, SaaS applications with critical user data
    - Rolling Upgrade: replace instances of services with new versions of the instances one/few at a time. For backend services with stateless instances or Kubernetes/ECS deployments with live traffic
- Canary Testing: continuous deployment analog of beta-testing a small set of users who will test the new release before a full rollout. For feature rollouts in mobile apps, changes to recommendation engines or search algorithms.
- A/B Testing: run two or more variations and experiment with real users and identify which alternative delivers the best business results. For UI changes in e-commerce sites, pricing models…

Common combinations

Combination	**Why It Works**
Microservices + Rolling Upgrade	Services are deployed independently, so rolling upgrades reduce risk while maintaining uptime.
Microservices + Canary Testing	Canary releases can be targeted to just one microservice, making fine-grained testing possible.
Blue/Green + A/B Testing	Route a portion of traffic to the Green environment to conduct A/B experiments safely.

Modifiability

About change and the cost and risk of making changes. What can change, what is the likelihood of the change, when and who makes it

Types of changes:

- Scalability: add resources (Horizontal: add more servers or logical units; Vertical: enhance physical resources)
- Variability: support different artifact versions
- Portability: run across different platforms
- Location independence: operate or relocate seamlessly across locations (physical or network locations, servers)

Modifiability General Scenario

Portion of Scenario	Possible Values
Source	End user, developer, system administrator, product line owner, system itself
Stimulus	A directive to add/delete/modify functionality, or change a quality attribute, platform, capacity, or technology, change location, new product in product line
Artifacts	Code, data, interfaces, components, resources, test cases, configurations, documentation
Environment	Runtime, compile time, build time, initiation time, design time
Response	One or more of the following: •Make modification •Test modification •Deploy modification
Response Measure	Cost in terms of: •number, size, complexity of affected artifacts •effort •elapsed time •money (direct outlay or opportunity cost) •extent to which this modification affects other functions or quality attributes •new defects introduced •how long it took the system to adapt

EXAMPLE

Source: movie theatre

Stimulus: add payment system

Environment: design time

Artefact: payment module

Response: modification made tested and deployed

Response measure: 4 hours without side effects

Modifiability Tactics

- Increase Cohesion
  - Split Module: break down large and complex modules into smaller ones to reduce future modifications cost.

    - Redistribute Responsibilities: consolidate responsibilities scattered between modules into one

- Reduce Coupling
    - Encapsulate: define a interface for a module while hiding internal details
    - Use an intermediary: to reduce direct coupling and dependencies between modules (performance is the tradeoff)
    - Abstract Common Services: consolidate similar services from two modules into a single more general implementation
    - Restrict Dependencies: limit the modules a given module interacts with or depends on
- Defer Binding (delaying decisions about values, components and behaviors as late as you can)
    - At compile or build time
        - Component replacements: swap implementations by changing the build (use one module for testing, another for production)
        - Compile-time parametrization: use compiler flags or macros to inject specific values
        - Aspects: Use aspect-oriented programming to inject behavior (logging, security) without modifying core logic

Patterns for Modifiability

- Client-Server Pattern: a server provides services to multiple distributed clients (useful to have like a core, the server). Clients discover the server's location and server responds with protocol and then they interact: client sends request and server processes request and responds. For web apps, enterprise backends, API-based services.
- Plug-in (Microkernel) Pattern: a core system (microkernel) provides basic services, and plug-ins can be added to extend behavior without altering the core. For IDEs, media players and browser extensions.
- Layers Pattern: system is divided into ordered layers, each with a specific responsibility, and layers only communicate downward. Makes it easy to swap or refactor layers independently. For enterprise applications, mobile apps…
- Publish-Subscribe Pattern: components communicate via asynchronous messages. Publisher sends messages, and a subscriber subscribes to and receives messages. Easy to add, remove or modify subscribers or publishers. For notification systems, event buses…

Common combinations

Combination	Why It Works
Client-Server + Plug-in	The server can expose extension points that clients interact with via plug-ins.
Layers + Plug-in	Each layer can be designed to load plug-ins (e.g., application layer loading new UI modules).
Publish-Subscribe + Client-Server	Clients and servers can interact normally, but use pub-sub for updates or event tracking.
Layers + Publish-Subscribe	Event-driven messaging within or between layers enables modular inter-layer communication.

Conclusion guide

Use **Layers** when you want **clear separation of responsibilities** and **stable interfaces**.

Use **Plug-in** when the system must be **easily extensible or support third-party modules**.

Use **Client-Server** when logic must be centralized but clients may vary (e.g., mobile, desktop).

Use **Publish-Subscribe** for **flexible communication** that scales well as functionality grows.

Energy Efficiency

Property of software to using no more energy than necessary to provide functionality. Effective resource utilization.

Energy Efficiency General Scenario

Portion of Scenario	Possible Values
Source	End user, manager, system administrator, automated agent
Stimulus	A request to conserve energy: Total usage, max instantaneous usage, average usage etc.
Artifact	Specific devices, servers, VMs, cluster, etc.
Environment	Runtime, connected, battery-powered, low-battery mode, power-conservation mode
Response	One or more of the following: Disable services, Deallocate runtime services, Change allocation of services to servers, Run services at lower consumption mode, Allocate/deallocate servers, Change levels of service, Change scheduling
Response Measure	Energy managed or saved in terms of: • Max/average kilowatt load of energy saved • Total kilowatt hours used • Time period during which the system must stay power on ... while still maintaining a required level of functionality and acceptable levels of other quality attributes

EXAMPLE

Source: manager

Stimulus: reduce total energy usage

Environment: runtime connected

Artifact: servers

Response: reduce level of service

Response measure: save 50% of energy and provide 5 sec user request latency and 10000 user simultaneously

Energy Efficiency Tactics

- Monitor Resources
    - Metering: real-time energy consumption data via sensors
    - Static classification: energy estimation based on computational resource characteristics
    - Dynamic classification: energy estimation using dynamic models like workload data
- Allocate Resources
    - Reduce Usage: adjusting settings like refresh rates, brightness, drive activity...
    - Discovery: identify energy efficient service with discovery tools

    - Schedule Resources: choose resources based on energy efficiency, cost and performance (if more expensive during the day, do it at night)
- Reduce Resource Demand
    - Even Management: limit, prioritize or reduce incoming events
    - Workload Reduction: decrease system load

Patterns for Energy Efficiency

- Sensor Fusion: use low-power sensors to decide if higher-power sensors are needed (use accelerometer before GPS). For fitness trackers, as an example.
- Kill Abnormal Tasks: track app energy use and stop high-energy tasks when necessary. Mobile OS killing background apps with excessive CPU use.
- Power Monitor: minimize active time and auto-disable unused system devices. For peripherals in low-power mode when idle, laptops turning off the screen…

Common combinations

Combination	**Why It Works**
Sensor Fusion + Power Monitor	Use sensor fusion to minimize sensor usage, and power monitor to shut down idle components
Kill Abnormal Tasks + Power Monitor	Kill runaway tasks while also reducing idle system energy use
All Three Together	Forms a **layered strategy**: prevent waste (kill), avoid unnecessary use (fusion), and power down what's not needed (monitor)

Conclusion guide

Use **Sensor Fusion** to avoid using expensive components unless absolutely necessary.

Use **Kill Abnormal Tasks** to catch and terminate **inefficient runtime behavior**.

Use **Power Monitor** to **proactively reduce baseline energy usage** of idle components.

Combine all three for a **comprehensive energy-aware system design**, especially in **mobile, IoT, or battery-limited** environments.

Performance

Ensures the system meets timing requirements. Timely response to events like interrupts, messages, user requests...

Performance General Scenario

Portion of Scenario	Possible Values
Source	Internal: One component may make a request of another, a timer may generate a notification External: User request, request from external system, data arriving form sensor or other system
Stimulus	Arrival of a periodic (predictable interval), stochastic event (according to some probability distribution), sporadic (neither periodic nor stochastic)
Artifact	Whole system or one components within the system.
Environment	Operational mode: normal, emergency, error correction, peak load, overload, degraded operation, other defined mode
Response	Returns response, returns error, generates no response, ignores request if overloaded, changes level of service, services higher-priority event, consumes resources
Response Measure	Latency, deadline, throughput, jitter, miss rate, resource use

EXERCISE/ EXAMPLE

Source: 5000 users / 200000 simultaneous users

Stimulus: try to get tickets inb a spanb of 5 seconds (stochastic) / book tickets for concert

Artifact: ticket booking system / system

Environment: peak load (big concert, for example) / normal operation

Response: process all requests / 5000 users get access to the system rest reject

Response measure: 5 seconds / average response time 1 second

Performance Tactics

- Control Resource Demand
    - Manage Work Requests: limit incoming requests to the system (arrival rates and/or sampling rate)
    - Limit Event Response: queue or discard events if CPU/queue size is exceeded
    - Prioritize Events: rank events by importance to ensure critical ones are addressed
    - Reduce Computational Overhead: minimize work required to handle each event

    - Bound Execution times: limit execution times to respond to events
    - Increase Resource Efficiency: improve algorithms to reduce latency (make code faster)
- Manage Resources
    - Increase Resources: more or faster CPUs, more memory, faster networks…
    - Introduce Concurrency: reduce blocked time by processing requests in parallel
    - Maintain Multiple Copies of Computations: distributing computations across multiple servers
    - Maintain Multiple Copies of Data: store data copies with varying access speeds
    - Bound Queue Sizes: limit number of queued arrivals
    - Schedule Resources: figure out efficient way of managing incoming events

Performance Patterns

- Service Mesh: move out all communication logic into a layer that uses sidecar proxys for reliable service to service communication. Security and communication rules are managed through a configurable control plane. Reduces communication overhead and increases reliability. For cloud-native microservices
- Load Balancer: (distribute traffic) intermediary component that manages messages from clients and directs them to the appropriate service instance. Serves as a single point for incoming messages and distributes requests to a server pool. Prevents overload, improves response time and optimizes resource use. For web applications with high traffic, container orchestration platforms.
- Throttling: (limit traffic) “throttler” intermediary that monitors requests and determines if they can be serviced or not to prevent overwhelming services. Protects performance during peak loads and ensures fairness or priority rules are applied. For API gateways, login systems, e-commerce platforms during peak sales…
- Map-Reduce: parallel processing model that breaks down large data processing tasks into independent smaller jobs (map) and the aggregate results (reduce). Reduces processing time for large datasets. For data analytics pipeline, log processing, machine learning preprocessing…

Common combinations

Combination	**Why It Works**
Load Balancer + Throttling	Balance incoming requests **and** ensure services aren’t overwhelmed.

Combination	Why It Works
Service Mesh + Throttling	Apply traffic policies and rate limiting transparently across microservices.
Map-Reduce + Load Balancer	Distribute **data-intensive jobs** efficiently across compute clusters.
Service Mesh + Load Balancer	Mesh handles **service-to-service**, balancer handles **external-to-service**.

Conclusion guide

Use Service Mesh when you need **reliable, secure, and observable service-to-service communication** in a distributed system.

Use Load Balancer to **distribute incoming client traffic** efficiently across instances and **maximize throughput**.

Use Throttling to **prevent system overload** by controlling the request rate, especially under burst or abusive traffic.

Use Map-Reduce when performing **large-scale data processing** that can be split and processed in **parallel**, especially in analytics and data engineering contexts.

Testability

Ease of identifying software faults through testing, makes system more isolated and predictable.

Testability General Scenario

Portion of Scenario	Possible Values
Source	Unit testers, Integration testers, System testers, Acceptance testers, End users. Either running tests manually or using automated testing tools
Stimulus	Test to: Validate system functions, Validate qualities, Discover emerging threats to quality
Environment	Tests executed due to: Completion of code increment (class, layer, service), Completed integration of subsystem, Completed implementation of whole system, Deployment of system into production environment, Delivery of system to customer, Testing schedule
Artifacts	The portion of system being tested (unit of code, components, services, entire system, test infrastructure)
Response	One or more of the following: Execute test suite & capture results; Capture activity that resulted in fault; Control & monitor state of system
Response Measure	One or more of the following: Effort to find a fault or class of faults, Effort to achieve a given percentage of state space coverage; Probability of fault being revealed by the next test; Time to perform tests; Effort to detect faults; Length of longest dependency chain in test; Length of time to prepare test environment; Reduction in risk exposure (size(loss) * prob(loss))

Testability Tactics

- Control and Observe System State
    - Specialized Interfaces
    - Record/Playback
    - Localiza State Storage
    - Abstract Data Sources
    - Sandbox
- Limit Complexity
    - Limit Structural Complexity
    - Limit Nondeterminism

Testability Patterns

- Dependency Injection Pattern Instead of a class creating or looking up its dependencies injected via constructor or method parameters. Easy to swap services with fakes during testing. For testing without relying on real database, API...
- Strategy Pattern: define algorithms or behaviors in an abstract interface and the actual implementation is selected at runtime. Easy to test different behaviors

independently. For payment systems to swap with real payment processors and fake ones.

- Intercepting Filter Pattern: Used to inject pre- and post-processing to a request or a response between a client and a service. Enables injection of test logic or inspection without modifying business code. For testing middleware (add filter that records or alters data during test runs)

Common combinations

Combination	**Why It Works**
Dependency Injection + Strategy	Inject different strategy implementations (e.g., mock vs real) to test isolated behaviors.
Intercepting Filter + Dependency Injection	Inject test-specific filters (e.g., logging, mocking responses) into the request pipeline.
Strategy + Intercepting Filter	Use filters to **control or observe** different strategy behaviors in integration tests.

Conclusion guide

Use Dependency Injection when you want to **decouple classes from their dependencies**, making **unit testing and mocking easier**.

Use Strategy Pattern to isolate and test **swappable behaviors or algorithms** independently, with minimal impact on the main code.

Use Intercepting Filter when you need to **inspect or manipulate requests and responses** in a test-friendly way without altering core logic.

Usability

How easily users accomplish tasks and the support provided by the system

Portion of Scenario	Possible Values
Source	End user, possibly in a specialized role (system or network administrator)
Stimulus	User wants to: Use system efficiently, Learn to use system, Minimize impact of errors, Adapt the system, or Configure the system
Environment	Runtime or System configuration time
Artifacts	Portion of system being stimulated: GUI, Command-line interface, Voice interface, Touch screen
Response	The system should: Provide user with the features needed, Anticipate the user's needs, Provide appropriate feedback to user
Response Measure	One or more of the following: Task time, Number of errors, Learning time, Ratio of learning time to task time, Number of tasks accomplished, User satisfaction, Gain of user knowledge, Ratio of successful operations to total operations, or Amount of time or data lost when an error occurs

response measure: user masters the system in x seconds or something like that, be careful with putting it like if it were performance.

EXAMPLE

Source: end user (first time visitor) /end user

Stimulous: learns to book a ticket /learn all funtionalitiy in gettick

Envirnment: runtime/ runtime

artifacts: the booking system /user-interface model

response: system should be intuitive and with clear simple buttons/ all functionality without errors

response measure: the user learns to use the system in 5 minutes /within 5 minutes

Usability Tactics

- Support User Initiative
    - Cancel: detect and terminate cancel requests, free resources, and notify components
    - Undo: retain system state data to restore earlier states upon request
    - Pause-Resume: temporarily free resources for reallocation
    - Aggregate: group objects for collective operations, reducing user effort.
- Support System Initiative
    - Mantain Task Model: identifies user context to assist with tasks.
    - Maintain User Model: tracks user knowledge and behavior for tailored responses.
    - Maintain System Model: keeps a system self-model to provide accurate feedback.

Usability Patterns

- Model View Controller: separate the model, logic and UI views, enabling independent updates. Model represents data and business logic, view displays data to the user, and controller handles user input, updates model or view accordingly. For GUI frameworks, UI and logic evolving independently…
- Observer: A subject maintains a list of observers and notifies them of changes. Observers update automatically when the subject's state changes. For real-time dashboards, event-driven UI, form validation…
- Memento: allows to capture and restore an object's state without exposing its internal structure. Originator (object whose state is saved), Memento (snapshot of state), Caretaker(request mementos when sending events to originator, stores history of mementos. For text editors, graphic tools, design environments…

Common combinations

Combination	**Why It Works**
MVC + Observer	Keeps views **automatically in sync** with model changes — perfect for reactive UIs.
MVC + Memento	Controller triggers **undo/redo** by calling caretaker; model restores state.
Observer + Memento	Observer tracks model changes, and caretaker stores states at key moments.

Conclusion guide

Use Model-View-Controller (MVC) to **decouple the user interface from the business logic**, allowing **clean updates and responsive UI design**.

Use Observer to enable **live synchronization** between data and views, ideal for **dynamic or real-time interfaces**.

Use Memento to support **undo/redo functionality**, improving **user control, error recovery, and trust** in interactive tools.

Integrability

Extent to which parts of a system or multiple systems can seamlessly integrate. Distance: Measure of alignment among interacting components, cooperation in achieving successful interactions (like a high-level modifiability)

Distances:

- Syntactic distance: agree on numbers and types of shared elements (parameters and data types)
- Data semantic distance: agree on data interpretation (altitude in meters)
- Behavioral semantic distance (agree on behavior across states and models)
- Temporal Distance: agreement on timing assumptions
- Resource distance: agreement on shared resource requirements

Integrability General Scenario

Portion of Scenario	Possible Values
Source	One or more of the following: Mission/system stakeholder, Component marketplace, Component vendor.
Stimulus	One of the following: Add new component, Integrate new version of existing component, Integrate existing components together with a new.
Artifact	One of the following: Entire system, Specific set of components, Component metadata, Component configuration.
Environment	One of the following: Development, Integration, Deployment, Runtime
Response	One or more of the following: Changes are {completed, integrated, tested, deployed}, Components in the new configuration are successfully and correctly exchanging info, Component in the new configuration are successfully collaborating, Components in the new configuration do not violate any resource limits.
Response Measure	One or more of the following: Cost in terms of one or more {number of components changed, percentage of code changed, lines of code changed, effort, money, calendar time}, Effects of other quality attribute response measures (to capture tradeoffs).

Integrability Tactics

- Limit Dependencies
    - Encapsulate: don't expose all internal functioning, use interfaces
    - Use an intermediary: go through common components to interact
    - Restrict communication paths: make rules for who you should interct with
    - Adhere to standards: enable integrability across platforms and vendors
    - Abstract common services: combine similar services under
- Adapt
    - Discover: catalog and translate between addresses and formats

    - Tailor interfaces: modify an interface by adding or hiding capabilities
    - Configure behavior: adjust a component's behavior at build time or runtime for different interface versions
- Coordinate
    - Orchestrate: who goes first, operating independently, coordinating
    - Manage Resources: govern access to computing resources, requiring components to request resources like threads or memory through it

Patterns for Integrability

- Wrapper: encapsulate components to translate the interface, hide elements… to match what the system expects. Reuse of code and adapting third-parties into the system.
- Bridges: facilitate communication between components with different interfaces by separating an abstraction from its implementation. Instead of tightly coupling a class to a specific implementation, it introduces an abstraction layer that holds a reference to an implementation interface. This allows you to mix and match abstractions and implementations without rewriting them. For file systems, cross-platform GUI libraries, database access layer…
  * **Bridge vs Adapter vs Wrapper**:
    - **Bridge**: Decouples *abstraction and implementation*, designed **from the start** to support variability.
    - **Adapter**: Converts an **existing interface** to match another (used to **retrofit** compatibility).
    - **Wrapper**: A general term for **encapsulating another component**, often to hide complexity.
- Mediators: combination of wrappers, bridges and runtime planning for interface translation and runtime coordination. Simplifies many-to-many relationships. For event buses, GUI systems with widgets, service coordinators in microservices
- Service-oriented architectural pattern: distributed independent components that provide or consume standalone services, often implemented in different languages or platforms. Loosely coupled, independently heavy, coarse-grained deployable services. For enterprise systems built in different stacks.
- Dynamic Discovery: allows clients to find and bind service providers at runtime instead of hardcoding endpoints or connections. Microservices using service registries, IoT systems where devices appear/disappear dynamically.

Common combinations

Combination	Why It Works
Wrapper + Bridge	Wrapper translates interface; Bridge decouples abstraction from implementation.
Mediator + SOA	Mediator coordinates service interactions in a loosely coupled service environment.
SOA + Dynamic Discovery	Enables **scalable, flexible service communication** across dynamic environments.
Wrapper + SOA	Wrap legacy components to expose them as modern web services.

Conclusion guide

Use Wrapper when integrating **legacy or third-party components** with different interfaces that must be adapted to your system.

Use Bridge when you need **abstraction between interface and implementation**, supporting **multiple interchangeable backends or platforms**.

Use Mediator when component interactions become **complex or many-to-many**, and central coordination improves modularity and clarity.

Use Service-Oriented Architecture when designing systems that require **modularity, platform independence, and reusable services**, especially in enterprise or cloud environments.

Use Dynamic Discovery when **components and services may change at runtime**, and **manual reconfiguration is not viable** (e.g., autoscaling, IoT, or cloud-native systems).

Safety

System's ability of preventing damage, injury, loss of life to actor in its environment.
Causes: omissions, commission, timing, system values, sequence errors, out of sequence

Safety General Scenario

Portion of Scenario	Possible Values
Source	Specific instance of a: Sensor, Software Component, Communication channel, Device (such as a clock)
Stimulus	Omission: Value never arrives, function never performed; Commission: function performed incorrectly, device produces spurious events or incorrect data; Specific instance of incorrect data: sensor reports incorrect data, software component produces incorrect results; Timing failure: Data arrives too late or too early, generated event occurs too late, too early or wrong rate, events occur in incorrect order
Artifact	Safety-critical portions of the system
Environment	System operating mode: Normal, degraded, manual, recovery
Response	Recognize unsafe state and one or more of the following: Avoid unsafe state, recover, continue in degraded or safe mode, shut down, switch to manual operation, switch to backup system, notify appropriate entities, log unsafe state
Response Measure	One or more of the following: Amount or % of entries into unsafe states that are avoided, Amount or % of unsafe states from which system can recover, Change in risk exposure: size (loss) x prob (loss), % or time the system can recover, Amount or % of time system is shut down, Elapsed time to enter and recover (from manual, safe or degraded mode)

Exercise / Example

Source: sensor/ proximity sensor

Stimulus: in a wrong position but still receives the signal of firing / detecs human withing working area

Artefact: arm of the robot / robot control software

Environment: normal /active operation

Response: not firing / recognise unsafe state, notify operator

Response measure: should happen every time it is unsafe? / never enters unsafe state

Safety Tactics

- Unsafe State Avoidance
    - Substitution: protection mechanisms
    - Predictive model: forecasts system health and resources, offering early warnings of potential issues
- Unsafe State Detection
    - Timeout: verifies if component operations meet timing constraints
    - Timestamp: identifies incorrect event sequences in distribute systems

    - Condition monitoring: checks process or device condition
    - Sanity Checking: validates how reasonable are the results
    - Comparison: detects unsafe states by comparing outputs from synchronized or replicated elements
- Containment
    - Redundancy: duplicate or something like that to allow system operation despite failure
    - Limit Consequences: mitigate effects of unsafe states (abort, degrade to critical functions…)
    - Barrier: prevents problem propagation
- Recovery
    - Rollback: revert to safe state
    - Repair State: create more fault-tolerant states to resume execution
    - Reconfiguration: recover by remapping logical architecture to working functional resources

Safety Patterns

- Redundant Sensors: duplicate sensors to ensure accurate safety assessments. For aircraft using multiple altimeters and accelerometers, autonomous vehicles, radars…
- Monitor- Actuator: verify commands with a monitoring component before sending to actuators (physical machines or things like that), to prevent unsafe or unintended actions. For robotic arms in industries, braking systems…
- Separated Safety: divide system into safety-critical and non-critical sections. This way a fault in a non-safety-critical function can't affect a safety-critical one. For drones with navigation and control are isolated, infotainment and braking system in a car independent…
- Design Assurance Levels: classification system that ranks system components based on their failure impact. A (Catastrophic), B (Hazardous), C (Mayor), D (Minor), E (No effect). For avionics software, rail systems…

Common combinations

Combination	**Why It Works**
Redundant Sensors + Monitor-Actuator	Redundancy ensures valid data; monitor verifies safety before action.

Combination	Why It Works
Separated Safety + DALs	Allows DAL A/B components to be isolated, reducing the complexity of the rest.
Monitor-Actuator + DALs	Enforces safety-critical behavior before actuation, with assurance level validation.

Conclusion guide

Use Redundant Sensors when **sensor failure could lead to unsafe actions**, especially in autonomous or high-reliability systems.

Use Monitor-Actuator to **validate safety before issuing control commands**, particularly in physical systems (robots, vehicles, machinery).

Use Separated Safety when designing mixed-criticality systems — **isolate safety-critical logic** from non-essential or user-facing components.

Use Design Assurance Levels (DALs) to guide **how rigorously each component must be developed and tested** based on the safety impact of its failure.

Security

Security: protect data and information from unauthorized access. Main characteristics Confidentiality, Integrity, Availabillity + Authentication, Nonrepudiation, Authorization

Security General Scenario

Portion of Scenario	Possible Values
Source	Human or another system which may have been previously identified (either correctly or incorrectly) or may be currently unknown. A human attacker may be from outside the organization or from inside the organization.
Stimulus	Unauthorized attempt is made to display data, change or delete data, access system services, change the system's behavior, or reduce availability.
Artifact	System services; data within the system; a component or resources of the system; data produced or consumed by the system
Environment	The system is either online or offline, connected to or disconnected from a network, behind a firewall or open to a network, fully operational, partially operational, or not operational
Response	Transactions are carried out in a fashion such that •data or services are protected from unauthorized access; •data or services are not being manipulated without authorization; •parties to a transaction are identified with assurance; •the parties to the transaction cannot repudiate their involvements; •the data, resources, and system services will be available for legitimate use. The system tracks activities within it by •recording access or modification, •recording attempts to access data, resources or services, •notifying appropriate entities (people or systems) when an apparent attack is occurring.
Response Measure	One or more of the following •how much of a system is compromised when a particular component or data value is compromised, •how much time passed before an attack was detected, •how many attacks were resisted, •how long does it take to recover from a successful attack, •how much data is vulnerable to a particular attack

Exercise / Example

Source: user/ hacker

Stimulus: tries to have access to other users' ticket/ access user database

Artifact: ticket/ user database

Environment: online and fully operational / online, normal operation

Response: unauthorized record (401 error) /attack rejected

Response measure: changing the qr of the ticket /reject 99,99% of attempts

Security Tactics

- Detect
    - Detect intrusion: analyze network traffic or service requests
    - Detect service Denial: compare incoming traffic patterns to known attack profiles
    - Verify Message Integrity: use checksums or hash for messages and files
    - Detect Message Delay: monitor message delivery times to identify suspicious timing anomalies

- Resist
    - Identify Actors: identify source of external inputs
    - Authenticate Actors: verify actors are who they say they are
    - Authorize Actors: ensure actors have as less access as possible
    - Limit access: restrict access to resources
    - Limit Exposure: reduce attack surface by minimizing access points
    - Encrypt Data: secure data and communications with encryption
    - Separate Entities: isolate systems using different networks, virtual machines or air gaps
    - Validate input: sanitize and verify incoming data
    - Change default credentials: require users to update default settings
- React
    - Revoke access: restrict resources if an attack is suspected, even limiting legitimate users
    - Restrict Login: limit access after repeated failed login attempts
    - Inform Actors: alert staff when something happens
- Recover
    - Audit: record user and system actions
    - Nonrepudiation: ensures senders can't deny sending messages nor receiving them

Patterns for Security

- Intercepting Validator: inserts wrapper between message source and destination to intercept and validate data (verify integrity). For API gateways or web-apps validating tokens or input data before processing
- Intrusion Prevention System: standalone element that monitors network traffic or system behavior to identify and analyze suspicious activity in real time. For enterprise networks to stop SQL injection, DoS attacks, embedded firewalls in IoT gateways…

Common combinations

(They should work fine together)

Conclusion guide

Use Intercepting Validator to **inspect and sanitize inputs or requests** before they enter critical parts of the system, especially in web or API interfaces.

Use Intrusion Prevention System (IPS) to **detect and automatically block suspicious behavior** at the **network or system level**, complementing application-level controls.