SOFTWARE ARCHITECTURE

Lecture 3: Architectural Tactics

Last Lecture

- System qualities
  - Availability, modifiability, performance, security, testability, usability
- Business qualities
  - Time to market, cost and benefit, projected lifetime of system, targeted market, rollout schedule, integration with legacy systems
- Architecture qualities
  - Conceptual integrity, correctness, completeness, buildability

This Lecture

- How to approach qualities?
- Qualities achieved via design decisions
- What design decisions needed for achieving a specific quality?

Tactics

- Tactic:
  - Design decision that influences the control of a quality attribute response
- Tactics can refine other tactics
  - Redundancy $\subseteq$ data redundancy, code redundancy
- Example:
  - One availability tactic: introduce redundancy
  - Implication: we also need synchronization of replicas
    - To ensure the redundant copy can be used if the original fails

Tactics vs Strategy

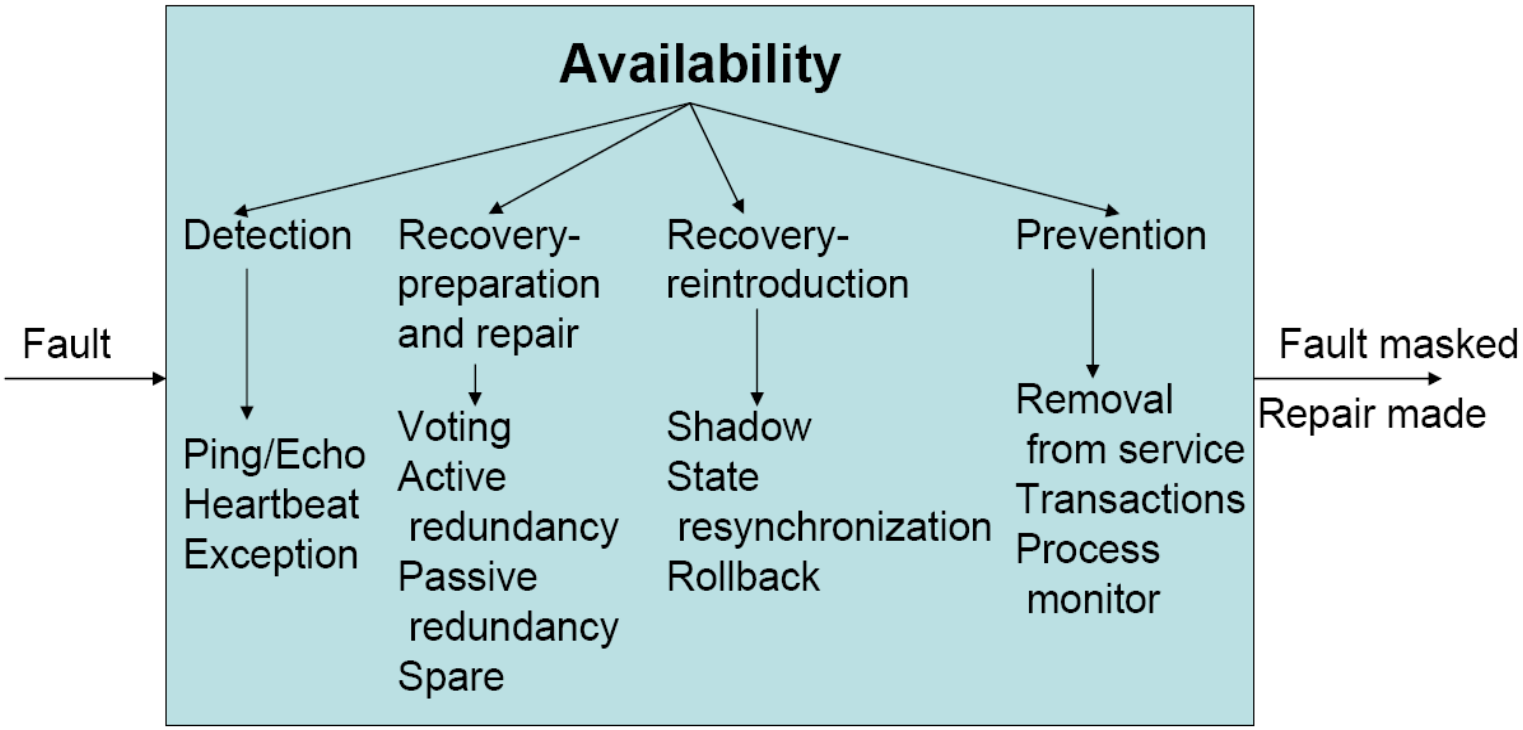
- OBAMA: They have done a brilliant job, and General Petraeus has done a brilliant job. But understand, that was a tactic designed to contain the damage of the previous four years of mismanagement of this war.
- MCCAIN: I'm afraid Senator Obama doesn't understand the difference between a tactic and a strategy.

*Barack Obama and John McCain during their debate at the University of Mississippi in Oxford, Mississippi, Sep. 27, 2008.

Availability

- Failure
  - Deviation from intended functional behavior
  - Observable by system users
- Failure vs fault
  - Fault: event which may cause a failure
- Availability tactics
  - Keep faults from becoming failures
  - Make possible repaires

Availability Tactics



Fault detection: Ping/Echo

- Comp. 1 issues a ”ping” to comp. 2
- Comp. 1 expects an ”echo” from comp. 2
- Answer within predefined time period
- Usable for a group of components
  - Mutually responsible for one task
- Usable for client/server
  - Tests the server and the communication path
- Hierarchy of fault detectors improves bandwidth usage

Fault detection: Heartbeat

- Comp. 1 emits a ”heartbeat” message periodically
- Comp. 2 listens for it
- If heartbeat fails
  - Comp. 1 assumed failed
  - Fault correcting comp. 3 is notified
- Heartbeat can also carry data

Fault detection: Exceptions

- Fault classes: omission, crash, timing, response
- When a fault class recognized, exception is raised
  - A fault consequently is recognized
- Exception handler
  - Executes in same process that introduced the exception
  - Typically does a semantic transformation of fault into a executable form

Fault recovery: Voting

- Processes running on redundant processors take equivalent input and compute output
- Output sent to voter
- Voter detects deviant behavior from a single processor => it fails it
- Method used to correct
  - Faulty operation of algorithm
  - Failure of a processor

Fault recovery: Active redundancy

- All redundant components respond to events in parallel => all in same state
- Response from only one comp used
- Downtime: switching time to another up-to-date component (ms)
- Used in client-server configurations (database systems)
  - Quick responses are important
- Synchronization
  - All messages to any redundant component sent to all redundant components

Fault recovery: Passive redundancy

- Primary component
  - responds to events
  - informs standby components of state updates they must make
- Fault occurs:
  - System checks if backup sufficiently fresh before resuming services
- Often used in control systems
- Periodical switchovers increase availability

Fault recovery: spare

- Standby spare computing platform configured to replace many different failed components
  - Must be rebooted to appropriate SW configuration
  - Have its state initialized when failure occurs
- Checkpoint of system state and state changes to persistent device periodically
- Downtime: minutes

Fault reintroduction: Shadow operation

- Previously failed component may be run in ”shadow” mode
  - For a while
  - To make sure it mimics the behavior of the working components
  - Before restoring it to service

Fault reintroduction: State re-synchronization

- Passive and active redundancy
  - Restored component upgrades its state before return to service
- Update depends on
  - Downtime
  - Size of update
  - Number of messages required for the update
    - One preferable; more lead to complicated SW

Fault reintroduction: Checkpoint/Rollback

- Checkpoint
  - Record of a consistent state
  - Created periodically or in response to specific events
- Useful when a system fails unusually, with detectably inconsistent state: system restored using
  - A previous checkpoint of a consistent state
  - Log of transactions occurred since

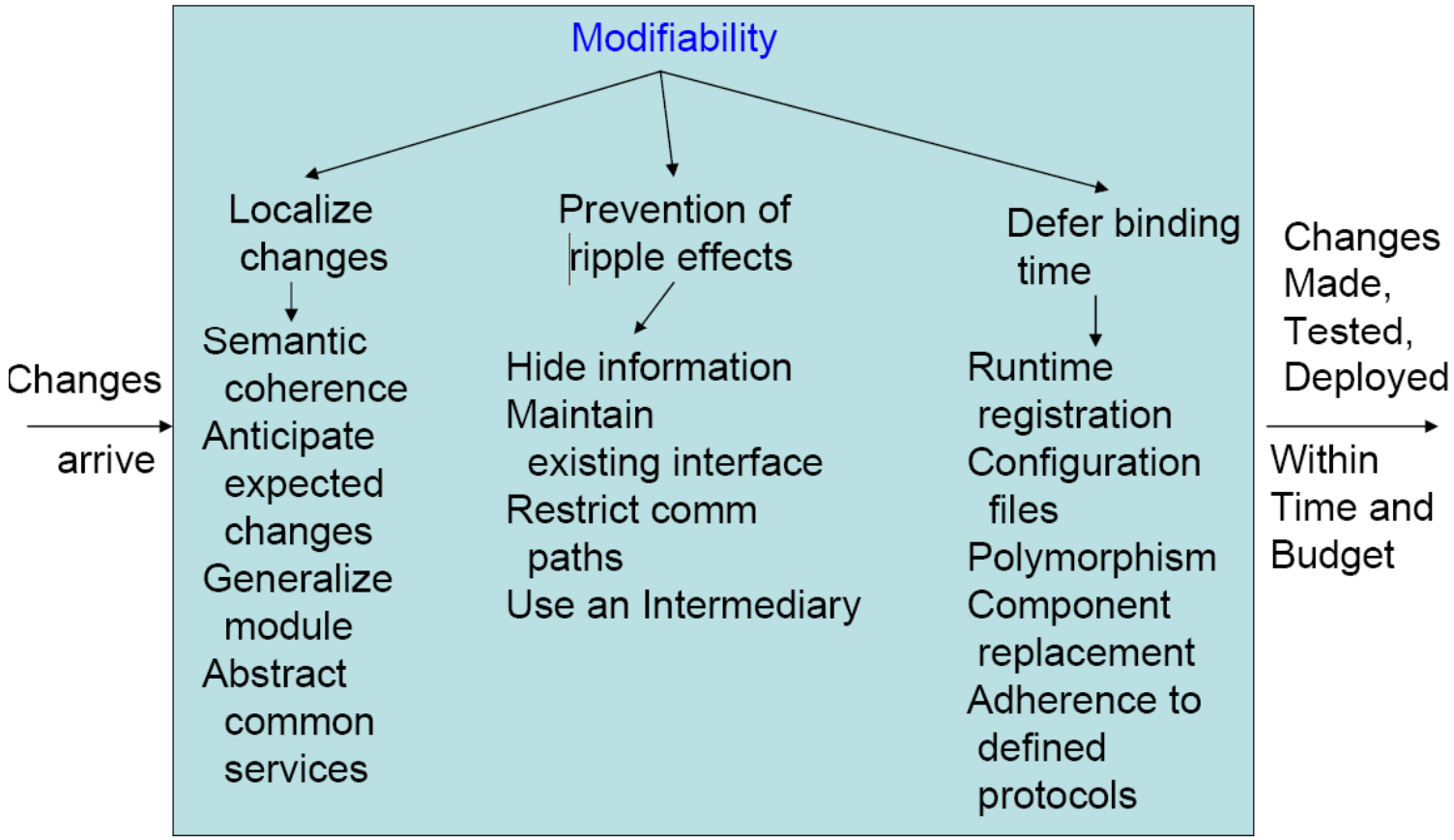
Fault prevention

- Removal from service
- Comp removed from operation to undergo some activities to prevent anticipated failures
- Ex.: rebooting comp to prevent memory leaks
- Automatic (design architecture) or manual (design system)
- Transactions
- Process monitor

Modifiability

- Goal: controlling time and cost to implement, test, modify and deploy changes
- Sets of tactics
  - Localize modifications
    - Reduce nr of modules affected by a change
  - Prevent ripple effects
    - Limiting modifications to localized modules
  - Defer binding time
    - Controlling deployment time and cost

Modifiability Tactics



Localize modifications: Maintain semantic coherence

- Semantic coherence: relationships among responsibilities in a module
- Goal: ensure all these responsibilities work together w/o excessive reliance on other modules
- Goal achievement: design modules with responsibilities in semantic coherence

Localize modifications: Abstract common services

- Subtactic of semantic coherence
- Provides common services through specialized modules
- Reuse and modifiability
  - Modifications to common services done only once rather than in every module using them
  - Modifications to modules using common services do not impact other users

Localize modifications: anticipate expected changes

- Considering the set of envisioned changes provides way to evaluate particular assignment of responsibilities
- Questions
  - For a change: does proposed decomposition limit the set of modules needing modifications?
  - Fundamentally diff. changes: do they affect the same modules?
- Goal: minimizing effects of changes

Localize modifications: generalize the module

- Generalize a module by making it compute a broader range of functions due to its input type
- Input → defining language for the module
  - Making constants input parameters
  - Implementing the module as an interpreter and making the input parameters be programs in that interpreter's language
- The more general the module
  - The most likely is that requested changes can be made by adjusting input language

Prevent ripple effects

- Localize modifications vs limit modifications to localized modules
  - There are modules directly affected
    - Whose responsibilities are adjusted to accomplish change
  - There are modules indirectly affected by a change
    - Whose responsibilities remain unchanged BUT implementation needs to be changed to accommodate the directly affected modules

Ripple effects

- Ripple effect from a modification
  - The necessity of making changes to modules not directly affected by it
  - This happens because said modules are SOMEHOW dependent on the modules directly dealing with the modification

Dependency types

- We assume
  - Module A changed to accomplish particular modification
  - Module B changed only because A changed
- There are several dependency types
  - Syntax, semantics, sequence, identity of interface, location of A, quality of service, existence of A, resource behavior of A

Syntax dependency

- Of data
  - B consumes data produced by A
  - Type and format of data in both A and B need to be consistent
- Of service
  - B invokes services of A
  - Signature of services produced by A need to be consistent with B's assumptions

Semantics dependency

- Of data
  - B consumes data produced by A
  - Semantics of data produced by A and consumed by B need to be consistent with B's assumptions
- Of service
  - B invokes services of A
  - Semantics of services produced by A need to be consistent with B's assumptions

Sequence dependency

- Of data
  - B consumes data produced by A
  - B must receive the data produced by A in a fixed sequence
- Of control
  - A must have executed previously within certain time constraints

Identity of interface of A

- A may have multiple interfaces
- B uses one of them
- For B to compile and execute correctly, the identity (name or handle) of the interface must be consistent with B's assumptions

Other dependencies

- Runtime location of A
  - Must be consistent with B's assumptions
- Quality of service/data provided by A
  - Properties involving the above quality must be consistent with B's assumptions
- Existence of A
  - For B to execute, A must exist
- Resource behavior of A
  - Must be consistent with B's assumptions

Tactics for ripple effect prevention

- Information hiding
- Maintain existing interfaces
- Restrict communication paths
- Use intermediary

Information hiding

- Decomposition of responsibilities into smaller pieces and choosing which information to make private and which public
- Public information available through specified interfaces
- Goal: isolate changes within one module and prevent changes from propagating to others
  - Oldest technique from preventing changes from propagating
  - Strongly related to ”anticipate expected changes” (it uses those changes as basis for decomposition)

Maintain existing interfaces

- B syntax-depends on A's interface
  - Maintaining the interface lets B stay unchanged
- Interface stability
  - Separating interface from implementation
- How to implement the tactic
  - Adding interfaces
  - Adding adapter
  - Providing a stub for A

Restrict communication paths

- Restrict number of modules with which the given module shares data
  - Reduce nr of modules that consume data produced by given module
  - Reduce nr of modules that produce data consumed by given module
- Reduced ripple effect
  - Data production/consumption introduces dependencies

Use an intermediary

- B dependent on A in other ways than semantically
  - Possible to introduce an intermediary to manage the dependency
  - Data (syntax)
  - Service (syntax)
  - Location of A
  - Existence of A

Defer binding time

- Decision can be bound into executing system at various times
- Binding at runtime
  - System has been prepared for that binding
  - All testing and distribution steps already completed
  - Supports end user/administrator making settings or providing input that affects behavior

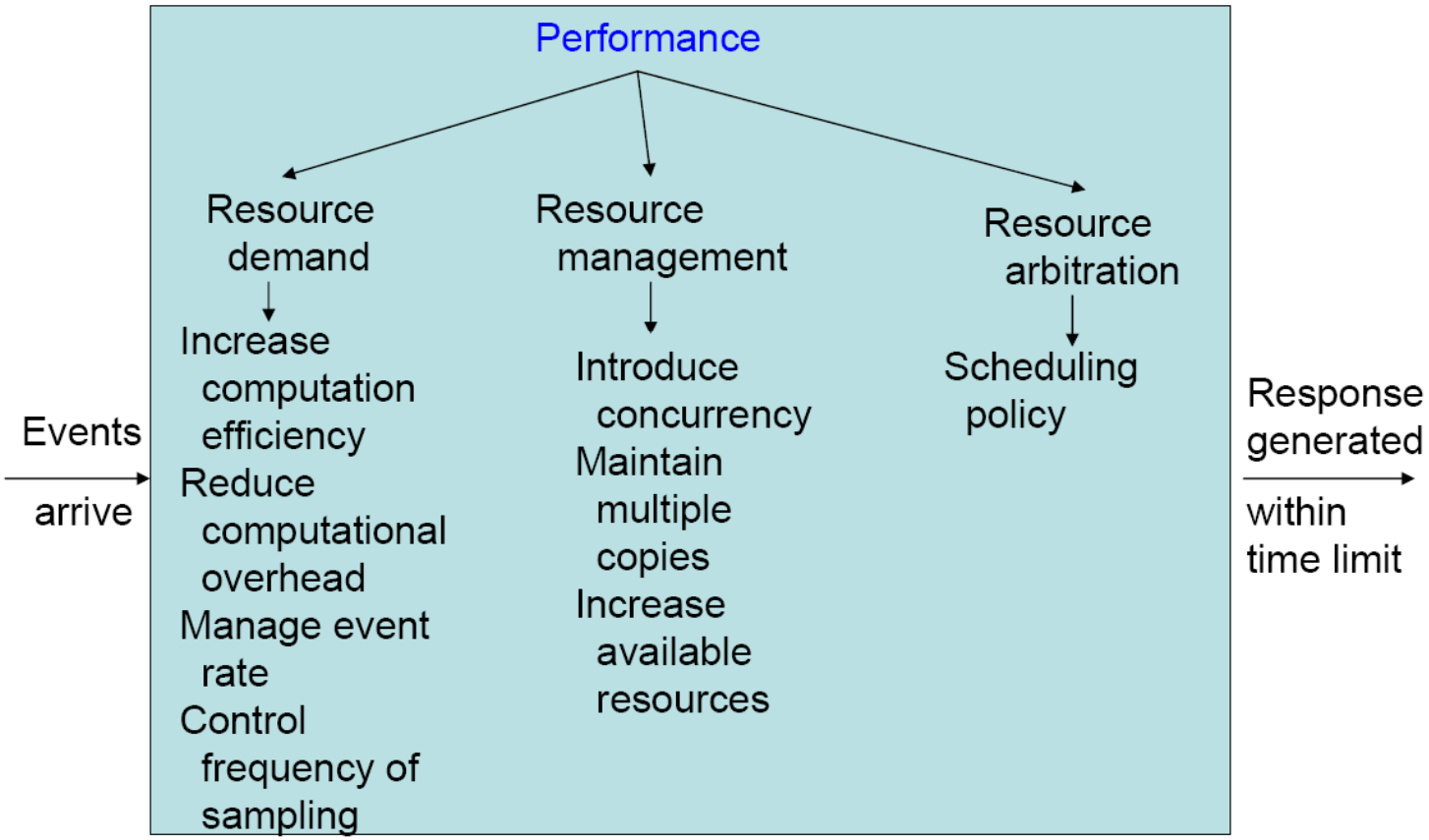
Tactics with impact at load/runtime

- Runtime registration
  - Plug-and-play operation, extra overhead to manage registration
- Configuration files - set parameters at start-up
- Polymorphism - late binding of method calls
- Component replacement – load time binding
- Adherence to defined protocols
  - Runtime binding of independent processes

Performance

- Goal: generate response to an event arriving at system within time constraint
- Event: single or stream
  - Message arrival, time expiration, significant state change etc
- Latency: time between the arrival of an event and the generation of a response to it
- Event arrives
  - System processes it or processing is blocked

Performance Tactics



Resource demand tactic

- Source of resource demand: event stream
- Demand characteristics
  - Time between events in resource stream (how often a request is made in a stream)
  - How much of a resource is consumed by each request
- Reducing latency tactic
  - Reduce required resources
  - Reduce nr of processed events

Reduce required resources

- Increase computational efficiency
  - Processing involves algorithms => improve algorithms
  - Resources may be traded for one another
- Reduce computational overhead
  - If no request for a resource => its processing needs are reduced
  - Intermediaries removed

Reduce no. of processed events

- Manage event rate
  - Reduce sampling frequency at which environmental variables are monitored
- Control frequency of samplings
  - If no control over arrival of externally generated events => queued requests can be sampled at a lower frequency (request loss)
- Bound execution times
  - Limit over how much execution time used for an event
- Bound queue sizes
  - Controls max nr of queued arrivals

Resource management

- Introduce concurrency
  - Process requests in parallel
    - Different event streams processed on different threads (create additional threads)
    - Load balancing
- Maintain multiple copies of data and computations
  - Caching and synchronization
- Increase available resources
  - Faster processors and networks, additional processors and memory

Resource arbitration

- Resource contention => resource must be scheduled
  - Architect's goal
  - Understand characteristics of each resource's use and choose compatible scheduling
- Scheduling policy
  - Priority assignment
  - Dispatching

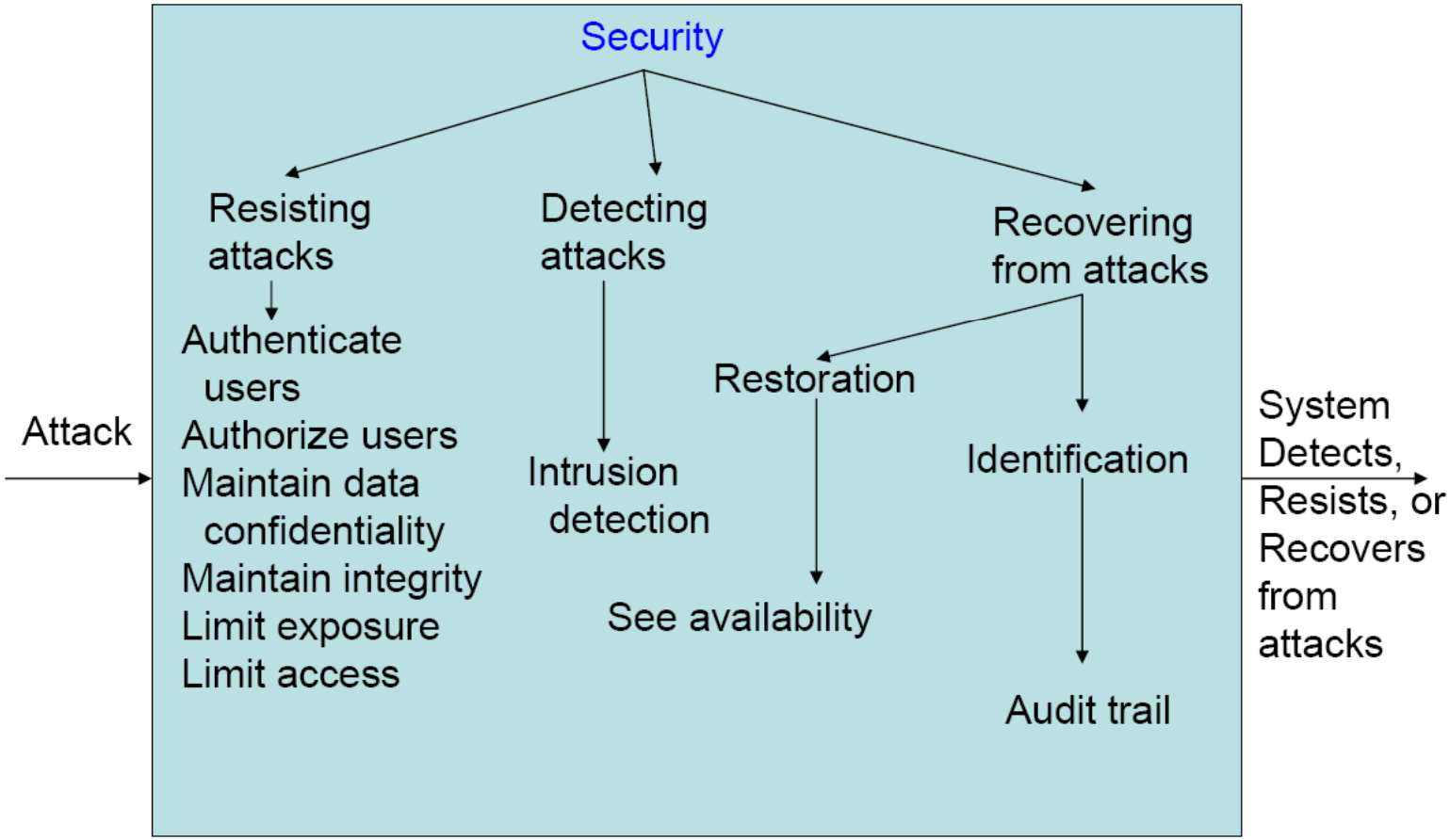
Scheduling policies

- First in/first out (FIFO)
- Fixed priority scheduling
- Dynamic priority scheduling
- Static scheduling

Security

- Goal: resisting attacks, detecting attacks, recovering from attacks
- House defense analogy
  - Door lock
  - Motion sensor
  - Insurance

Security Tactics



Resisting attacks

- Authenticate users
- Authorize users
- Maintain data confidentiality
- Maintain integrity
- Limit exposure
- Limit access

Detecting attacks

- Intrusion detection system
  - Compares network traffic patterns to database
  - Misuse => pattern compared to historical patterns of known attacks
  - Anomaly => pattern compared to historical baseline of itself
  - Filtering
    - Protocol, TCP flags, payload size, addresses, port numbers

Intrusion detectors

- Sensor to detect attacks
- Managers for sensor fusion
- Databases for storing events for later analysis
- Tools for offline reporting and analysis
- Control console
  - Analyst can modify intrusion detection actions

Recovering from attacks

- Tactics for restoring state
  - Recovering a consistent state from an inconsistent one: availability tactics
  - Redundant copies of system admin data
    - Passwords, access control lists, domain name services, user profile data: special attention
- Tactics for attacker identification
  - For preventive or punitive purposes
  - Maintain audit trail

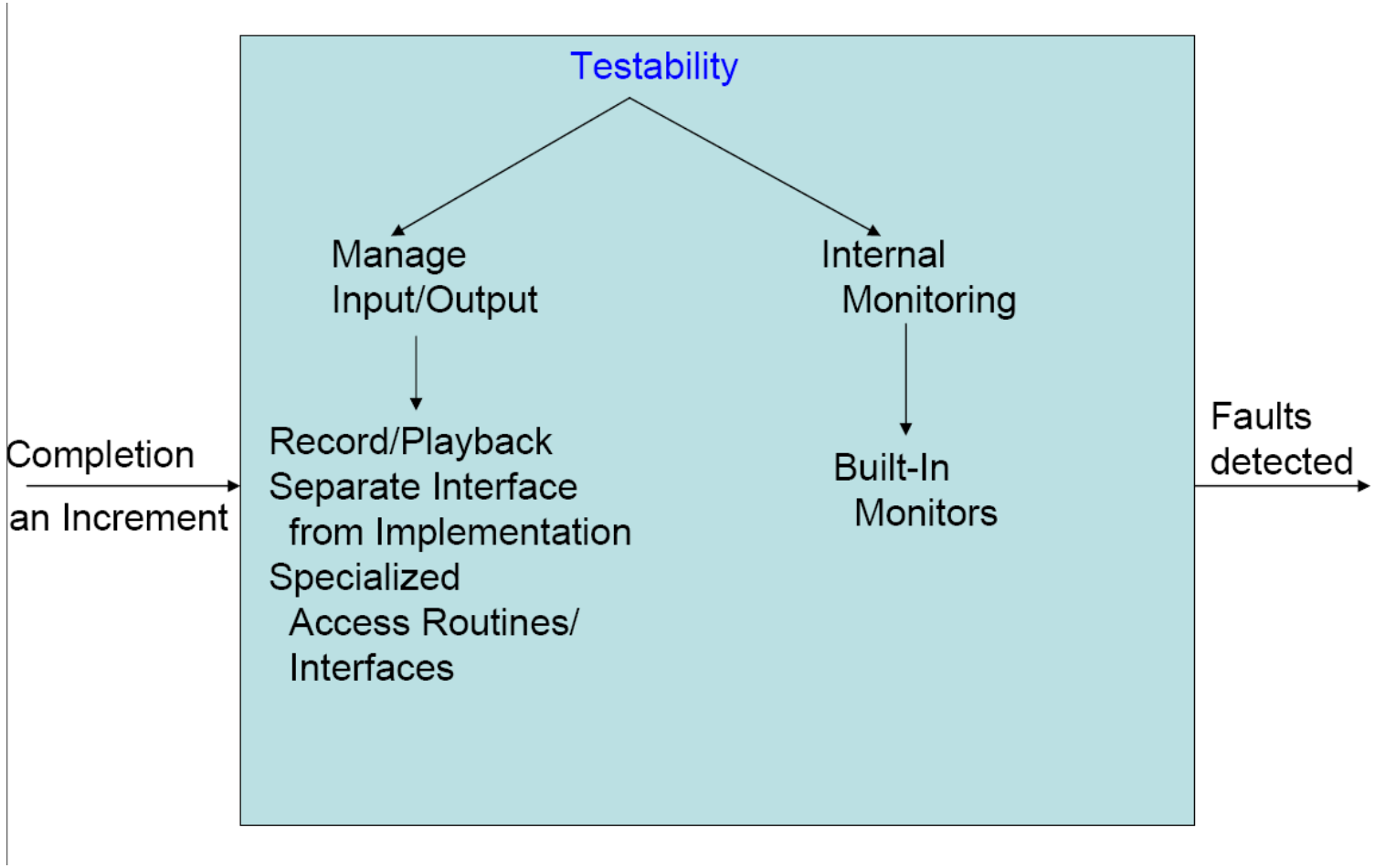
Audit trail

- Copy of each transaction applied to data in the system + identifying information
- Can be used to
  - Trace the actions of an attacker
  - Support non-repudiation
    - Provides evidence that a particular request was made
  - Support system recovery
- Often attack targets
  - Should be maintained in trusted fashion

Testability

- Goal: allow for easier testing when some increment of software development is completed
- Enhancing testability not so mature but very valuable
  - 40% of system development
- Testing a running system (not designs)
- Test harness
  - SW that provides input to the SW being tested and captures the output

Testability Tactics



I/O Tactics: Record/Playback

- Refers to
  - Capturing info crossing an interface
  - Using it as input to the test harness
- Info crossing an interface at normal operation
  - Output from one component, input to another
  - Saved in repository
    - Allows test input for one component
    - Gives test output for later comparisons

I/O Tactics: Interface vs. Implementation

- Separating interface from implementation
  - Allows substitution of implementations for various testing purposes
    - Stubbing implementations let system be tested without the component being stubbed
    - Substituting a specialized component lets the component being replaced to act as test harness for the remainder of the system

I/O Tactics: specialize access routes/interfaces

- Having specialized testing interfaces
  - Captures/specifies variable values for components
    - Via test harness
    - Independently from normal execution
- Specialized access routes/interfaces
  - Should be kept separate from required functions
- Hierarchy of test interfaces
  - Test cases can be applied at any arch. level

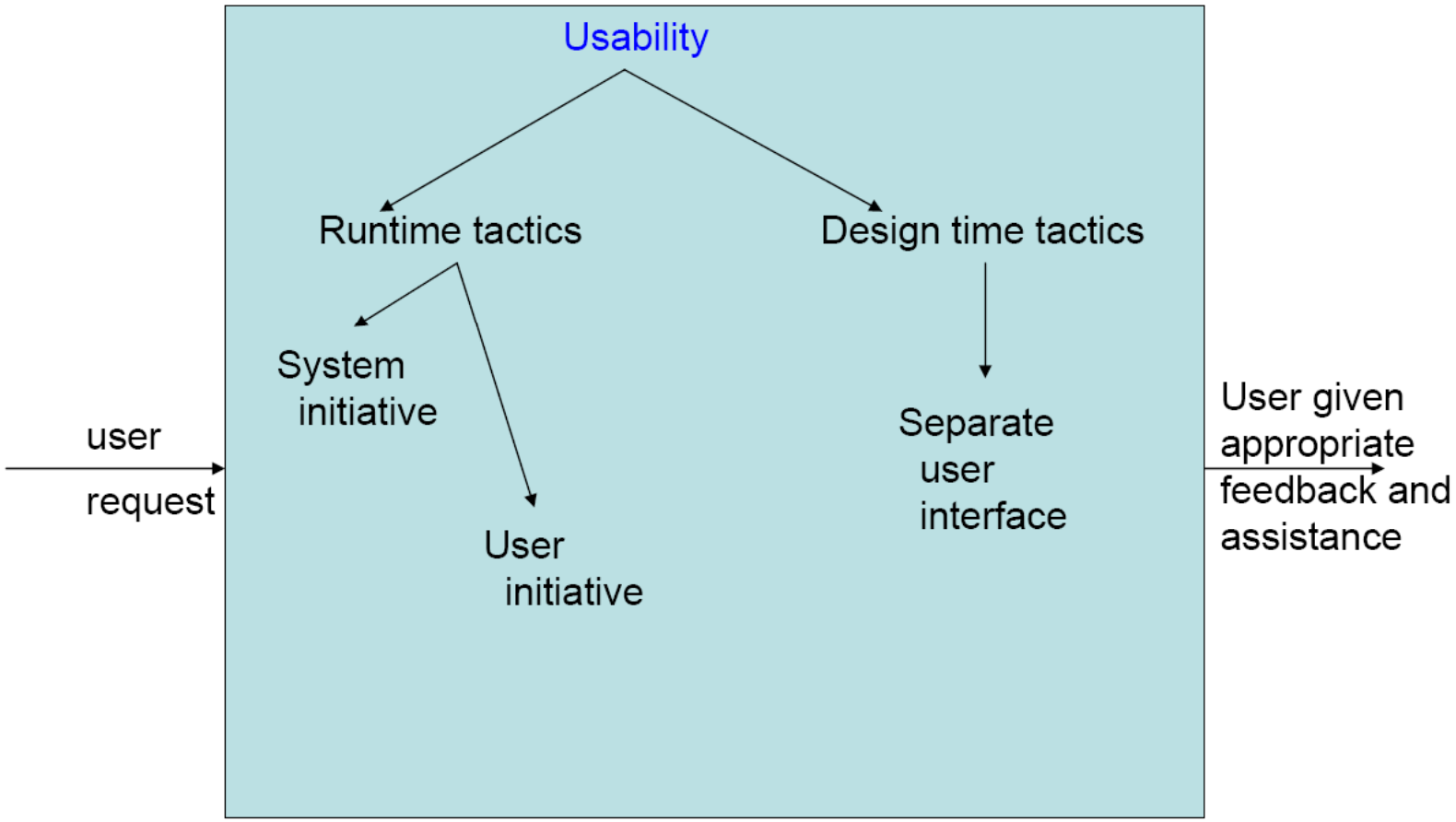
Internal monitoring tactic

- Built-in monitors
  - Component can maintain state, performance load, capacity, security, etc accessible through interfaces
    - Permanent interface or temporarily introduced for testing
  - Record events when monitoring states activated
    - Additional testing cost/effort

Usability

- Usability concerns
  - How easy can a user accomplish desired task
  - What is the system support for the user
- Tactics
  - Runtime: support user during system execution
  - Design time: support interface developer
    - Iterative nature of interface design
    - Related to modifiability tactics

Runtime Usability Tactics



Runtime tactics

- User feedback as to what is the system doing
- Providing user with ability to issue usability commands
  - Cancel
  - Undo
  - Aggregate
  - Show multiple views