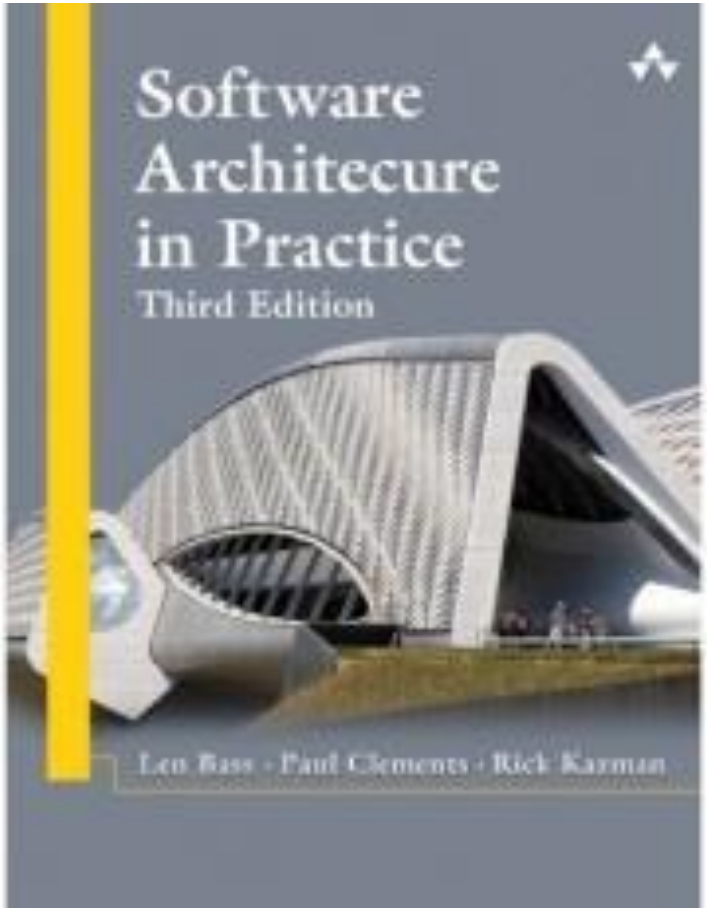
SE 4352
Software Architecture and Design

Fall 2018

Module 12

Chapter 5


Software
Architecure
in Practice
Third Edition
Len Bass · Paul Clements · Rick Kazman

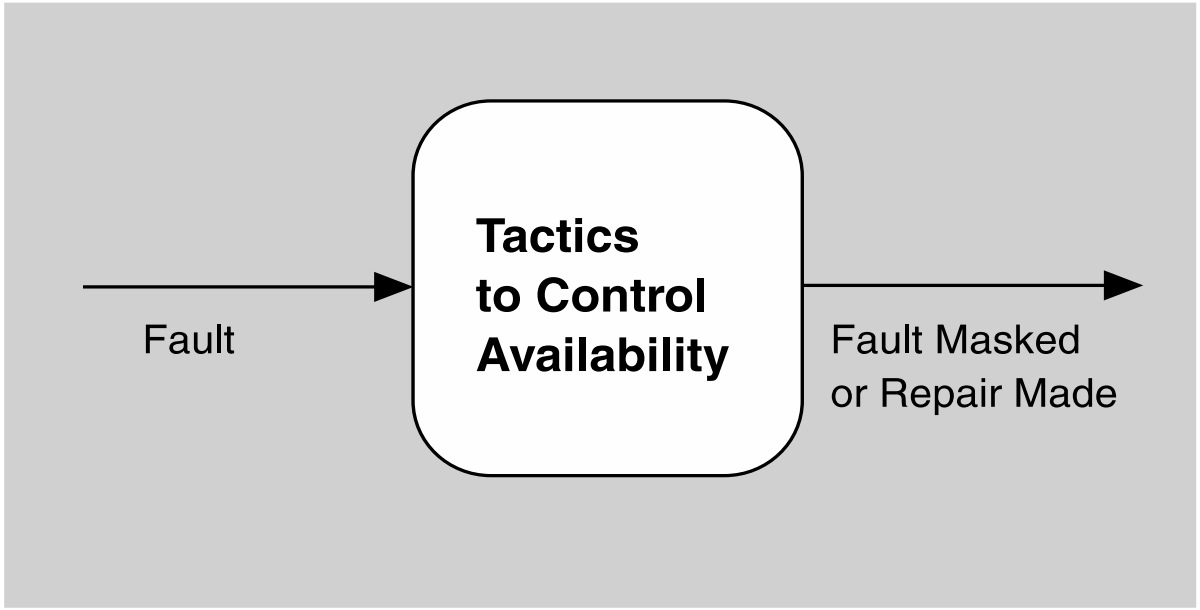
What is Availability?

- Availability refers to a property of software that it is there and ready to carry out its task when you need it to be.
- This is a broad perspective and encompasses what is normally called reliability.
- Availability builds on reliability by adding the notion of recovery (repair).
- Fundamentally, availability is about minimizing service outage time by mitigating faults.

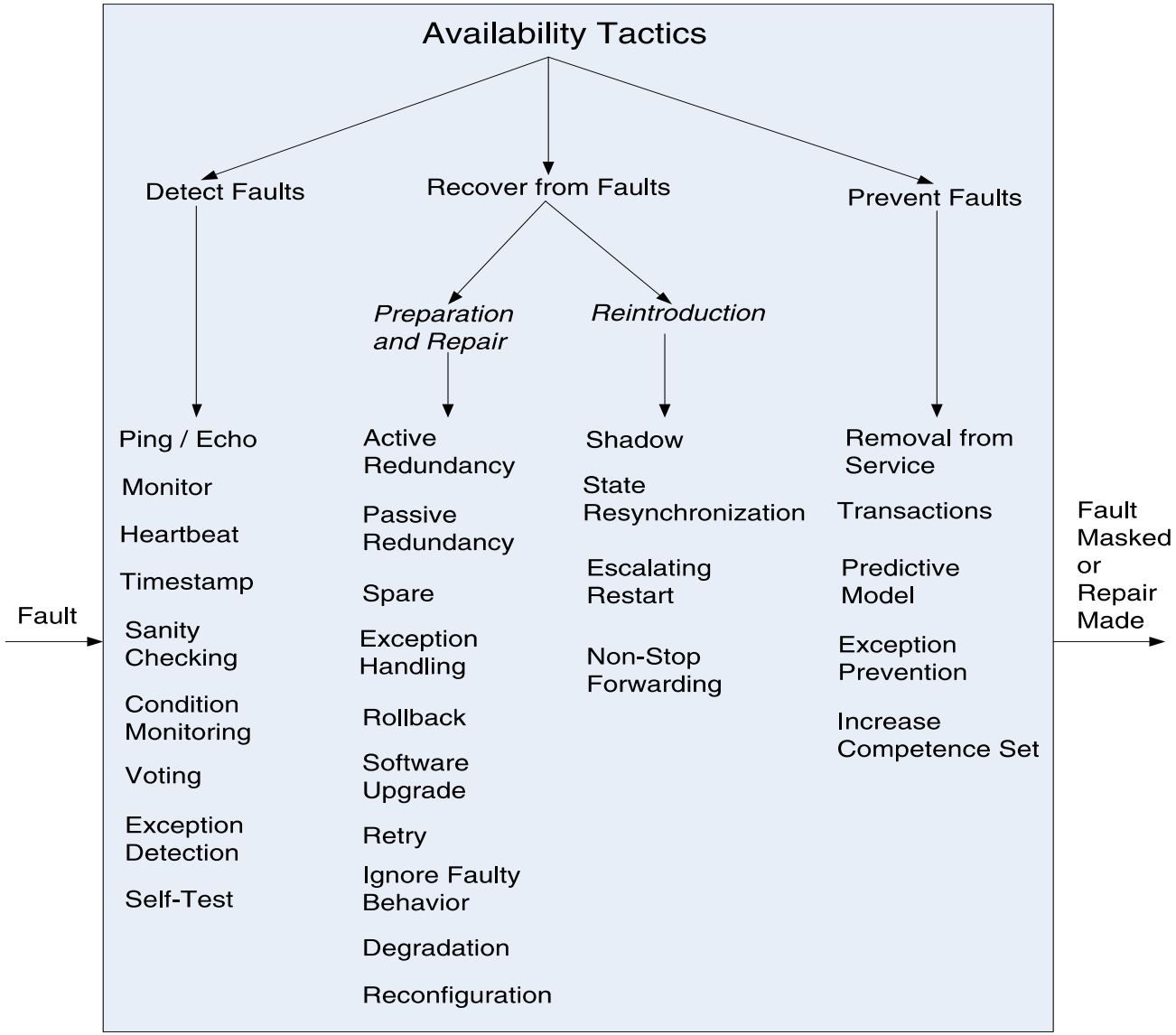
Goal of Availability Tactics

- A failure occurs when the system no longer delivers a service consistent with its specification
  - this failure is observable by the system's actors.
- A fault (or combination of faults) has the potential to cause a failure.
- Availability tactics enable a system to endure faults so that services remain compliant with their specifications.
- The tactics keep faults from becoming failures or at least bound the effects of the fault and make repair possible.

Goal of Availability Tactics



Availability Tactics



Detect Faults

- Ping/echo: asynchronous request/response message pair exchanged between nodes, used to determine reachability and the round-trip delay through the associated network path.
- Monitor: a component used to monitor the state of health of other parts of the system. A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack.
- Heartbeat: a periodic message exchange between a system monitor and a process being monitored.

Detect Faults

- Timestamp: used to detect incorrect sequences of events, primarily in distributed message-passing systems.
- Sanity Checking: checks the validity or reasonableness of a component's operations or outputs; typically based on a knowledge of the internal design, the state of the system, or the nature of the information under scrutiny.
- Condition Monitoring: checking conditions in a process or device, or validating assumptions made during the design.

Detect Faults

- Voting: to check that replicated components are producing the same results. Comes in various flavors: replication, functional redundancy, analytic redundancy.
- Exception Detection: detection of a system condition that alters the normal flow of execution, e.g. system exception, parameter fence, parameter typing, timeout.
- Self-test: procedure for a component to test itself for correct operation.

Recover from Faults (Preparation & Repair)

- Active Redundancy (hot spare): all nodes in a *protection group* receive and process identical inputs in parallel, allowing redundant spare(s) to maintain synchronous state with the active node(s).
  - A protection group is a group of nodes where one or more nodes are “active,” with the remainder serving as redundant spares.
- Passive Redundancy (warm spare): only the active members of the protection group process input traffic; one of their duties is to provide the redundant spare(s) with periodic state updates.
- Spare (cold spare): redundant spares of a protection group remain out of service until a fail-over occurs, at which point a power-on-reset procedure is initiated on the redundant spare prior to its being placed in service.

Recover from Faults (Preparation & Repair)

- Exception Handling: dealing with the exception by reporting it or handling it, potentially masking the fault by correcting the cause of the exception and retrying.
- Rollback: revert to a previous known good state, referred to as the “rollback line”.
- Software Upgrade: in-service upgrades to executable code images in a non-service-affecting manner.

Recover from Faults (Preparation & Repair)

- Retry: where a failure is transient retrying the operation may lead to success.
- Ignore Faulty Behavior: ignoring messages sent from a source when it is determined that those messages are spurious.
- Degradation: maintains the most critical system functions in the presence of component failures, dropping less critical functions.
- Reconfiguration: reassigning responsibilities to the resources left functioning, while maintaining as much functionality as possible.

Recover from Faults (Reintroduction)

- Shadow: operating a previously failed or in-service upgraded component in a “shadow mode” for a predefined time prior to reverting the component back to an active role.
- State Resynchronization: partner to active redundancy and passive redundancy where state information is sent from active to standby components.
- Escalating Restart: recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected.
- Non-stop Forwarding: functionality is split into supervisory and data. If a supervisor fails, a router continues forwarding packets along known routes while protocol information is recovered and validated.

Prevent Faults

- Removal From Service: temporarily placing a system component in an out-of-service state for the purpose of mitigating potential system failures
- Transactions: bundling state updates so that asynchronous messages exchanged between distributed components are *atomic*, *consistent*, *isolated*, and *durable*.
- Predictive Model: monitor the state of health of a process to ensure that the system is operating within nominal parameters; take corrective action when conditions are detected that are predictive of likely future faults.

Prevent Faults

- Exception Prevention: preventing system exceptions from occurring by masking a fault, or preventing it via smart pointers, abstract data types, wrappers.
- Increase Competence Set: designing a component to handle more cases—faults—as part of its normal operation.

Design Guidelines

- Allocation of Responsibilities
- Coordination between components
- Data
- Mapping of Components
- Resource Allocation

Design Checklist for Availability

Allocation of Responsibilities	**Determine the system responsibilities that need to be highly available. Ensure that additional responsibilities have been allocated to detect an omission, crash, incorrect timing, or incorrect response.** **Ensure that there are responsibilities to:** • **log the fault** • **notify appropriate entities (people or systems)** • **disable source of events causing the fault** • **be temporarily unavailable** • **fix or mask the fault/failure** • **operate in a degraded mode**

Design Checklist for Availability

Coordination Model	**Determine the system responsibilities that need to be highly available. With respect to those responsibilities** • **Ensure that coordination mechanisms can detect an omission, crash, incorrect timing, or incorrect response. Consider, e.g., whether guaranteed delivery is necessary. Will the coordination work under degraded communication?** • **Ensure that coordination mechanisms enable the logging of the fault, notification of appropriate entities, disabling of the source of the events causing the fault, fixing or masking the fault, or operating in a degraded mode** • **Ensure that the coordination model supports the replacement of the artifacts (processors, communications channels, persistent storage, and processes). E.g., does replacement of a server allow the system to continue to operate?** • **Determine if the coordination will work under conditions of degraded communication, at startup/shutdown, in repair mode, or under overloaded operation. E.g., how much lost information can the coordination model withstand and with what consequences?**

Design Checklist for Availability

Data Model	**Determine which portions of the system need to be highly available. Within those portions, determine which data abstractions could cause a fault of omission, a crash, incorrect timing behavior, or an incorrect response.** **For those data abstractions, operations, and properties, ensure that they can be disabled, be temporarily unavailable, or be fixed or masked in the event of a fault.** **E.g., ensure that write requests are cached if a server is temporarily unavailable and performed when the server is returned to service.**

Design Checklist for Availability

Mapping Among Architectural Elements	**Determine which artifacts (processors, communication channels, storage, processes) may produce a fault: omission, crash, incorrect timing, or incorrect response.** **Ensure that the mapping (or re-mapping) of architectural elements is flexible enough to permit the recovery from the fault. This may involve a consideration of** **. which processes on failed processors need to be re-assigned at runtime** **. which processors, data stores, or communication channels can be activated or re-assigned at runtime** **. how data on failed processors or storage can be served by replacement units** **. how quickly the system can be re-installed based on the units of delivery provided** **. how to (re-) assign runtime elements to processors, communication channels, and data stores** **When employing tactics that depend on redundancy of functionality, the mapping from modules to redundant components is important. E.g., it is possible to write a module that contains code appropriate for both the active and back-up components in a protection group.**

Design Checklist for Availability

Resource Management	**Determine what critical resources are necessary to continue operating in the presence of a fault: omission, crash, incorrect timing, or incorrect response. Ensure there are sufficient remaining resources in the event of a fault to log the fault; notify appropriate entities (people or systems); disable source of events causing the fault; fix or mask the fault/failure; operate normally, in startup, shutdown, repair mode, degraded operation, and overloaded operation.** **Determine the availability time for critical resources, what critical resources must be available during specified time intervals, time intervals during which the critical resources may be in a degraded mode, and repair time for critical resources. Ensure that the critical resources are available during these time intervals.** **For example, ensure that input queues are large enough to buffer anticipated messages if a server fails so that the messages are not permanently lost.**

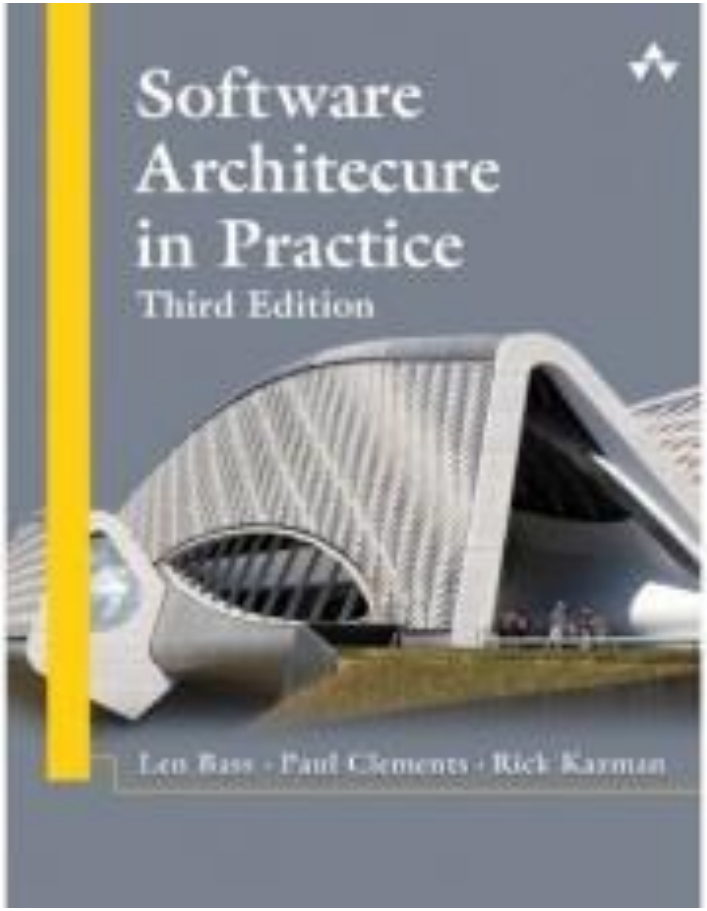
Summary

- Availability refers to the ability of the system to be available for use when a fault occurs.
- The fault must be recognized (or prevented) and then the system must respond.
- The response will depend on the criticality of the application and the type of fault
  - can range from “ignore it” to “keep on going as if it didn’t occur.”

Summary

- Tactics for availability are categorized into detect faults, recover from faults and prevent faults.
- Detection tactics depend on detecting signs of life from various components.
- Recovery tactics are retrying an operation or maintaining redundant data or computations.
- Prevention tactics depend on removing elements from service or limiting the scope of faults.
- All availability tactics involve the coordination model.

Chapter 6


Software
Architecure
in Practice
Third Edition
Len Bass · Paul Clements · Rick Kazman

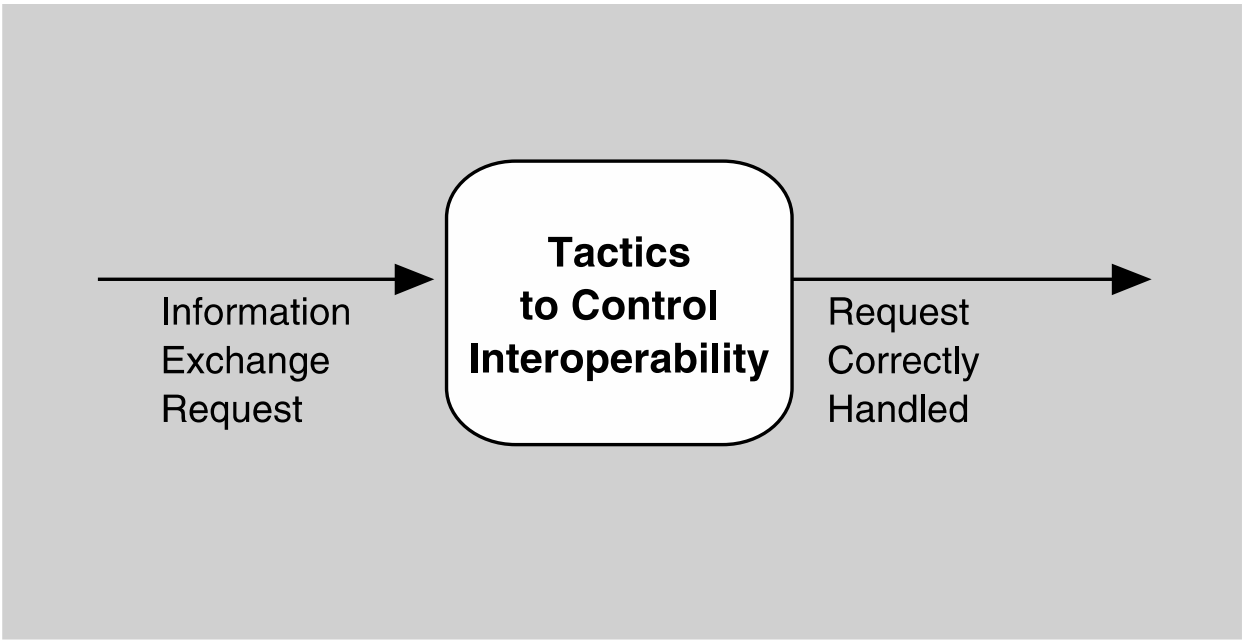
What is Interoperability?

- Interoperability is about the degree to which two or more systems can usefully exchange meaningful information.
- Like all quality attributes, interoperability is not a yes-or-no proposition but has shades of meaning.

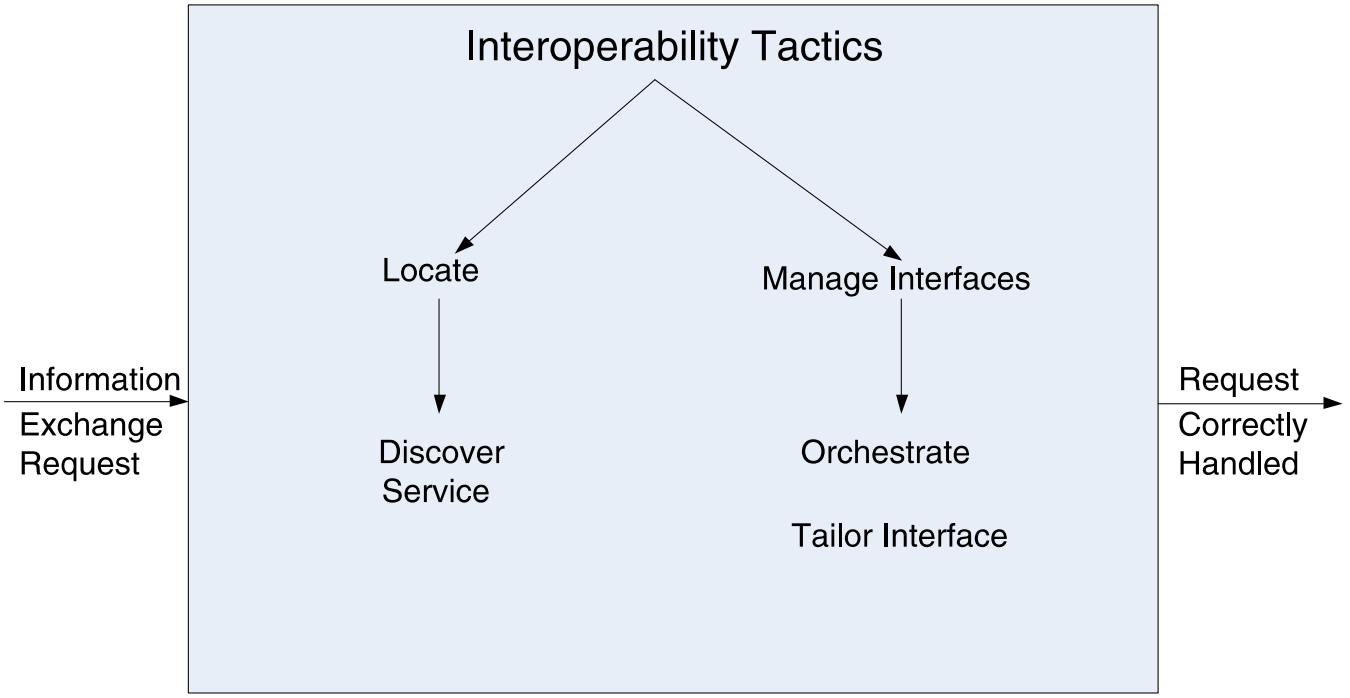
Goal of Interoperability Tactics

- For two or more systems to usefully exchange information they must
  - Know about each other. That is the purpose behind the locate tactics.
  - Exchange information in a semantically meaningful fashion. That is the purpose behind the manage interfaces tactics. Two aspects of the exchange are
    - Provide services in the correct sequence
    - Modify information produced by one actor to a form acceptable to the second actor.

Goal of Interoperability Tactics



Interoperability Tactics



Locate

- Discover service: Locate a service through searching a known directory service. There may be multiple levels of indirection in this location process – i.e. a known location points to another location that in turn can be searched for the service.

Manage Interfaces

- Orchestrate: uses a control mechanism to coordinate, manage and sequence the invocation of services. Orchestration is used when systems must interact in a complex fashion to accomplish a complex task.
- Tailor Interface: add or remove capabilities to an interface such as translation, buffering, or data-smoothing.

Design Checklist for Interoperability

Allocation of Responsibilities	**Determine which of your system responsibilities will need to interoperate with other systems.** **Ensure that responsibilities have been allocated to detect a request to interoperate with known or unknown external systems** **Ensure that responsibilities have been allocated to** • **accept the request** • **exchange information** • **reject the request** • **notify appropriate entities (people or systems)** • **log the request (for interoperability in an untrusted environment, logging for non-repudiation is essential)**

Design Checklist for Interoperability

Coordination Model	**Ensure that the coordination mechanisms can meet the critical quality attribute requirements. Considerations for performance include:** • **Volume of traffic on the network both created by the systems under your control and generated by systems not under your control.** • **Timeliness of the messages being sent by your systems** • **Currency of the messages being sent by your systems** • **Jitter of the messages arrival times.** **Ensure that all of the systems under your control make assumptions about protocols and underlying networks that are consistent with the systems not under your control.**

Design Checklist for Interoperability

Data Model	**Determine the syntax and semantics of the major data abstractions that may be exchanged among interoperating systems.** **Ensure that these major data abstractions are consistent with data from the interoperating systems. (If your system's data model is confidential and must not be made public, you may have to apply transformations to and from the data abstractions of systems with which yours interoperates.)**

Design Checklist for Interoperability

Mapping Among Architectural Elements	**For interoperability, the critical mapping is that of components to processors. Beyond the necessity of making sure that components that communicate externally are hosted on processors that can reach the network, the primary considerations deal with meeting the security, availability, and performance requirements for the communication.** **These will be dealt with in their respective chapters.**

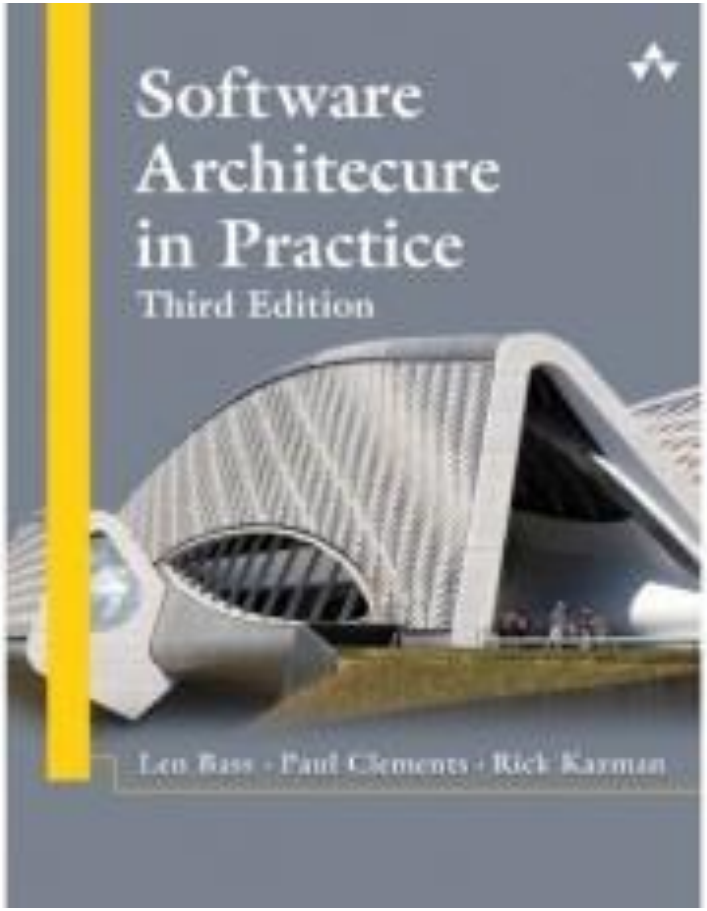
Design Checklist for Interoperability

Resource Management	**Ensure that interoperation with another system (accepting a request and/or rejecting a request) can never exhaust critical system resources (e.g., can a flood of such requests cause service to be denied to legitimate users?).** **Ensure that the resource load imposed by the communication requirements of interoperation is acceptable.** **Ensure that if interoperation requires that resources be shared among the participating systems, an adequate arbitration policy is in place.**

Summary

- Interoperability refers to the ability of systems to usefully exchange information.
- Achieving interoperability involves the relevant systems locating each other and then managing the interfaces so that they can exchange information.

Chapter 7


Software
Architecure
in Practice
Third Edition
Len Bass · Paul Clements · Rick Kazman

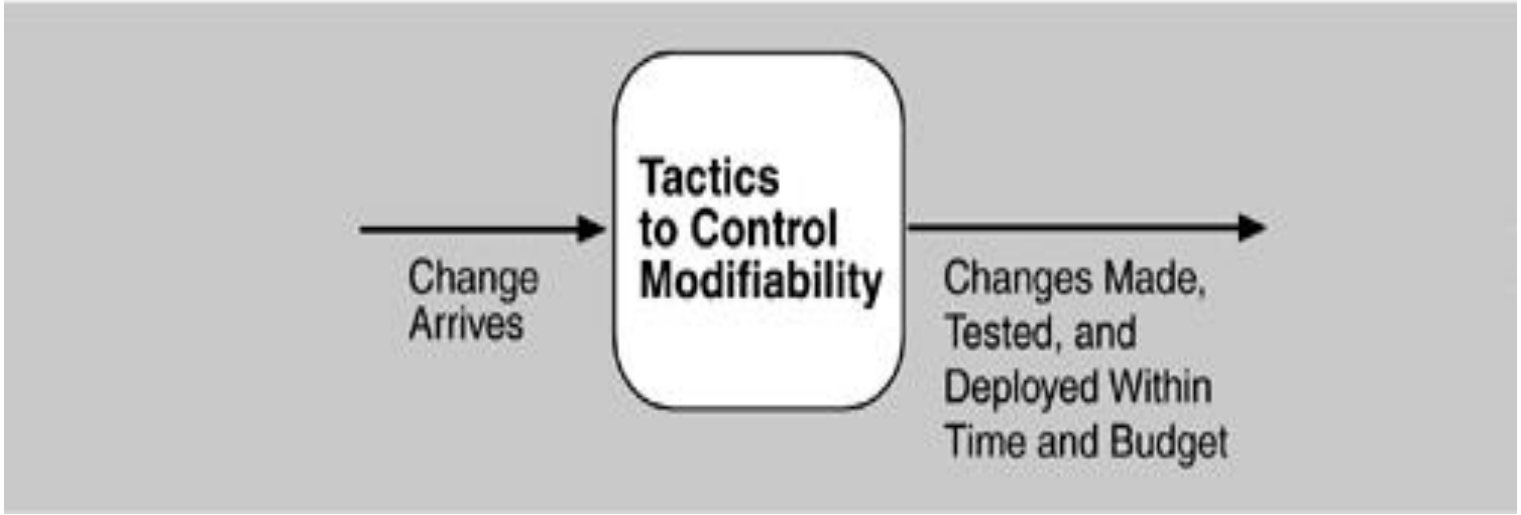
What is Modifiability?

- Modifiability is about change and our interest in it is in the cost and risk of making changes.
- To plan for modifiability, an architect has to consider three questions:
  - What can change?
  - What is the likelihood of the change?
  - When is the change made and who makes it?

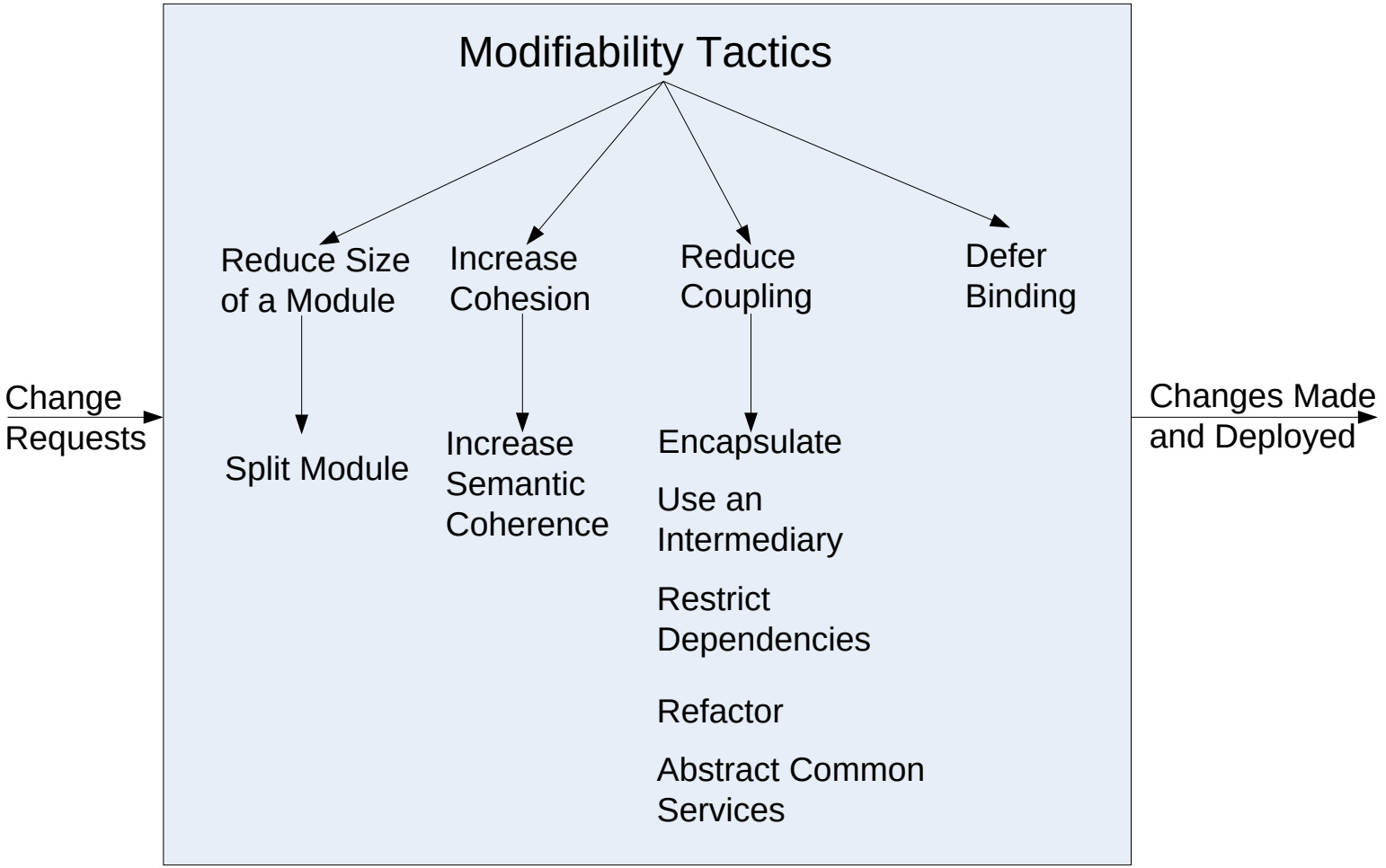
Goal of Modifiability Tactics

- Tactics to control modifiability have as their goal controlling the complexity of making changes, as well as the time and cost to make changes.

Goal of Modifiability Tactics



Modifiability Tactics



Reduce Size of a Module

- 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

- 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.

Reduce Coupling

- 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.

Design Checklist for Modifiability

Allocation of Responsibilities	**Determine which changes or categories of changes are likely to occur through consideration of changes in technical, legal, social, business, and customer forces. For each potential change or category of changes** . **Determine the responsibilities that would need to be added, modified, or deleted to make the change.** . **Determine what other responsibilities are impacted by the change.** . **Determine an allocation of responsibilities to modules that places, as much as possible, responsibilities that will be changed (or impacted by the change) together in the same module, and places responsibilities that will be changed at different times in separate modules.**

Design Checklist for Modifiability

Coordination Model	**Determine which functionality or quality attribute can change at runtime and how this affects coordination; for example, will the information being communicated change at run-time, or will the communication protocol change at run-time? If so, ensure that such changes affect a small number set of modules.** **Determine which devices, protocols, and communication paths used for coordination are likely to change. For those devices, protocols, and communication paths, ensure that the impact of changes will be limited to a small set of modules.** **For those elements for which modifiability is a concern, use a coordination model that reduces coupling such as publish/subscribe, defers bindings such as enterprise service bus, or restricts dependencies such as broadcast.**

Design Checklist for Modifiability

Data Model	**Determine which changes (or categories of changes) to the data abstractions, their operations, or their properties are likely to occur. Also determine which changes or categories of changes to these data abstractions will involve their creation, initialization, persistence, manipulation, translation, or destruction.** **For each change or category of change, determine if the changes will be made by an end user, system administrator, or developer. For those changes made by an end user or administrator, ensure that the necessary attributes are visible to that user and that the user has the correct privileges to modify the data, its operations, or its properties.** **For each potential change or category of change** • **determine which data abstractions need to be added, modified, or deleted** • **determine whether there would be any changes to the creation, initialization, persistence, manipulation, translation, or destruction of these data abstractions** • **determine which other data abstractions are impacted by the change. For these additional abstractions, determine whether the impact would be on their operations, properties, creation, initialization, persistence, manipulation, translation, or destruction.** • **ensure an allocation of data abstractions that minimizes the number and severity of modifications to the abstractions by the potential changes** **Design your data model so that items allocated to each element of the data model are likely to change together.**

Design Checklist for Modifiability

Mapping Among Architectural Elements	**Determine if it is desirable to change the way in which functionality is mapped to computational elements (e.g. processes, threads, processors) at runtime, compile time, design time, or build time.** **Determine the extent of modifications necessary to accommodate the addition, deletion, or modification of a function or a quality attribute. This might involve a determination of, for example:** **. execution dependencies** **. assignment of data to databases** **. assignment of runtime elements to processes, threads, or processors** **Ensure that such changes are performed with mechanisms that utilize deferred binding of mapping decisions.**

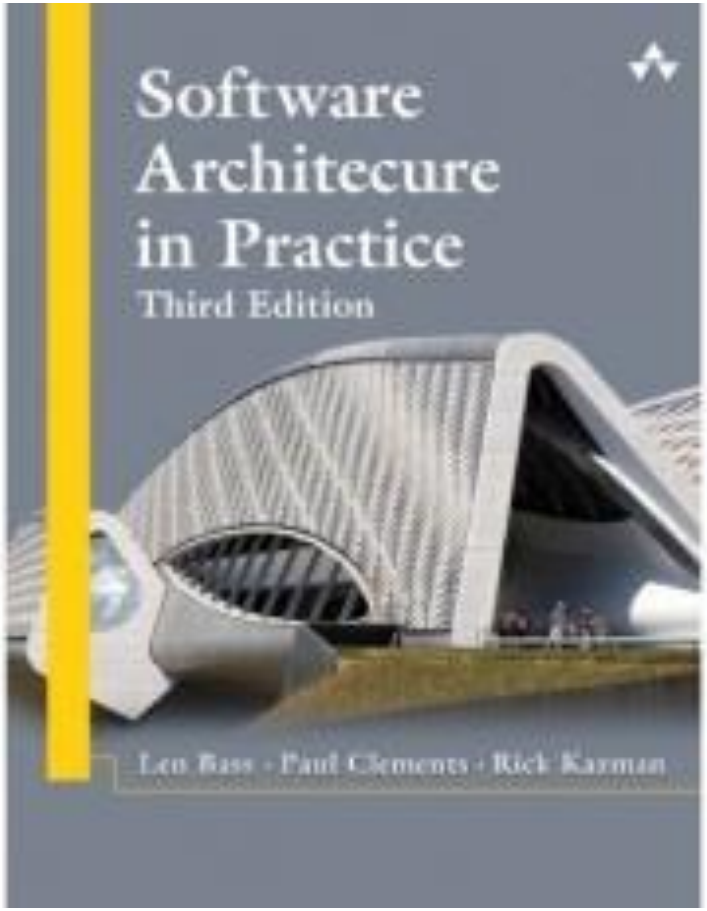
Design Checklist for Modifiability

Resource Management	**Determine how the addition, deletion, or modification of a responsibility or quality attribute will affect resource usage. This involves, for example,** **. determining what changes might introduce new resources or remove old ones or affect existing resource usage.** **. determining what resource limits will change and how** **Ensure that the resources after the modification are sufficient to meet the system requirements.** **Encapsulate all resource managers and ensure that the policies implemented by those resource managers utilize are themselves encapsulated and bindings are deferred to the extent possible.**

Summary

- Modifiability deals with change and the cost in time or money of making a change, including the extent to which this modification affects other functions or quality attributes.
- Tactics to reduce the cost of making a change include making modules smaller, increasing cohesion, and reducing coupling. Deferring binding will also reduce the cost of making a change.

Chapter 8


Software
Architecure
in Practice
Third Edition
Len Bass · Paul Clements · Rick Kazman

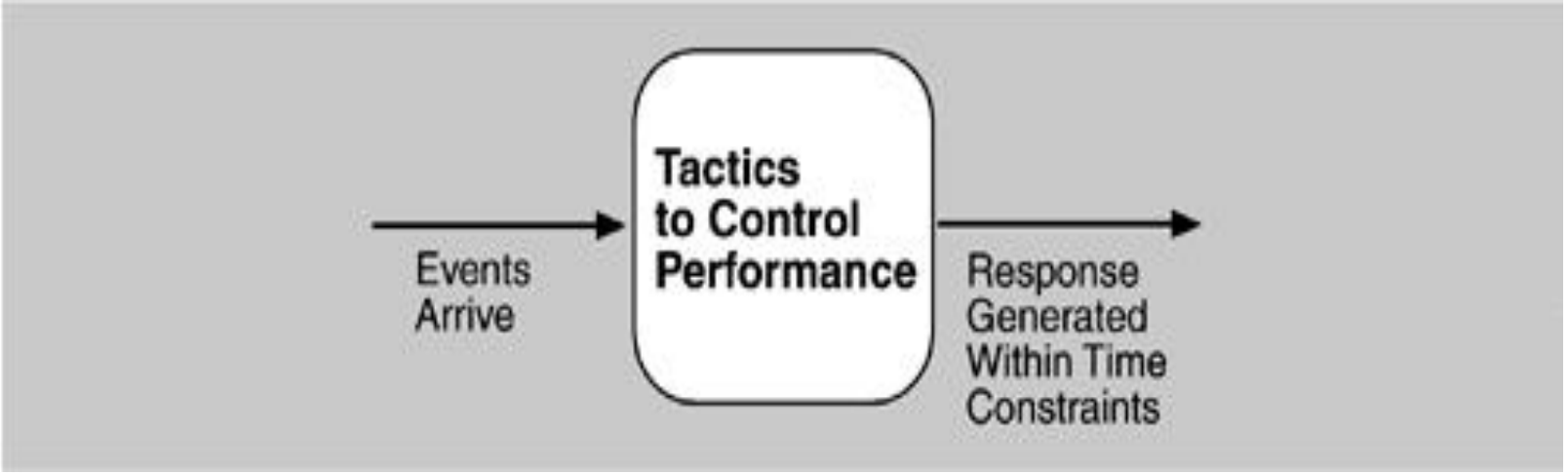
What is Performance?

- Performance is about time and the software system's ability to meet timing requirements.
- When events occur – interrupts, messages, requests from users or other systems, or clock events marking the passage of time – the system, or some element of the system, must respond to them in time.
- Characterizing the events that can occur (and when they can occur) and the system or element's time-based response to those events is the essence is discussing performance.

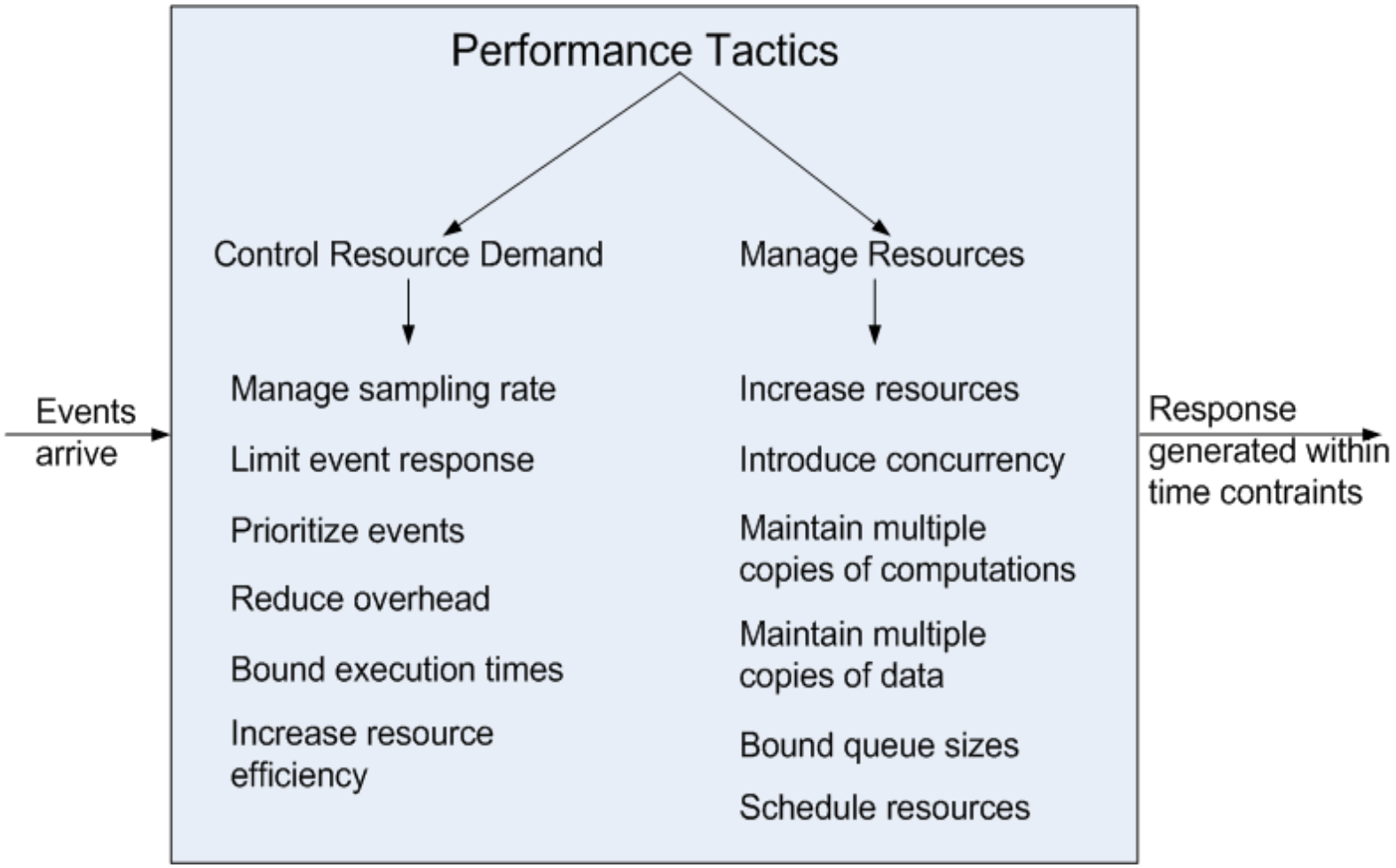
Goal of Performance Tactics

- Tactics to control Performance have as their goal to generate a response to an event arriving at the system within some time-based constraint.

Goal of Performance Tactics



Performance Tactics



Control Resource Demand

- Manage Sampling Rate: If it is possible to reduce the sampling frequency at which a stream of data is captured, then demand can be reduced, typically with some loss of fidelity.
- Limit Event Response: process events only up to a set maximum rate, thereby ensuring more predictable processing when the events are actually processed.
- Prioritize Events: If not all events are equally important, you can impose a priority scheme that ranks events according to how important it is to service them.

Control Resource Demand

- Reduce Overhead: The use of intermediaries (important for modifiability) increases the resources consumed in processing an event stream; removing them improves latency.
- Bound Execution Times: Place a limit on how much execution time is used to respond to an event.
- Increase Resource Efficiency: Improving the algorithms used in critical areas will decrease latency.

Manage Resources

- Increase Resources: Faster processors, additional processors, additional memory, and faster networks all have the potential for reducing latency.
- Increase Concurrency: If requests can be processed in parallel, the blocked time can be reduced. Concurrency can be introduced by processing different streams of events on different threads or by creating additional threads to process different sets of activities.
- Maintain Multiple Copies of Computations: The purpose of replicas is to reduce the contention that would occur if all computations took place on a single server.

Manage Resources

- Maintain Multiple Copies of Data: keeping copies of data (possibly one a subset of the other) on storage with different access speeds.
- Bound Queue Sizes: control the maximum number of queued arrivals and consequently the resources used to process the arrivals.
- Schedule Resources: When there is contention for a resource, the resource must be scheduled.

Design Checklist for Performance

Allocation of Responsi-bilities	**Determine the system's responsibilities that will involve heavy loading, have time-critical response requirements, are heavily used, or impact portions of the system where heavy loads or time critical events occur.** **For those responsibilities, identify** • **processing requirements of each responsibility and determine whether they may cause bottlenecks** • **additional responsibilities to recognize and process requests appropriately including** • **Responsibilities that result from a thread of control crossing process or processor boundaries.** • **Responsibilities to manage the threads of control — allocation and de-allocation of threads, maintaining thread pools, and so forth.** • **Responsibilities for scheduling shared resources or managing performance-related artifacts such as queues, buffers, and caches.** **For the responsibilities and resources you identified, ensure that the required performance response can be met (perhaps by building a performance model to help in the evaluation).**

Design Checklist for Performance

Coordination Model	**Determine the elements of the system that must coordinate with each other—directly or indirectly—and choose communication and coordination mechanisms that** • **supports any introduced concurrency (for example, is it thread-safe?), event priorization, or scheduling strategy** • **ensures that the required performance response can be delivered** • **can capture periodic, stochastic, or sporadic event arrivals, as needed** • **have the appropriate properties of the communication mechanisms, for example, stateful, stateless, synchronous, asynchronous, guaranteed delivery, throughput, or latency.**

Design Checklist for Performance

Data Model	**Determine those portions of the data model that will be heavily loaded, have time critical response requirements, are heavily used, or impact portions of the system where heavy loads or time critical events occur.** **For those data abstractions, 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.**

Design Checklist for Performance

Mapping Among Architectural Elements	**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.** **Determine where introducing concurrency (that is, allocating a piece of functionality to two or more copies of a component running simultaneously) is feasible and has a significant positive effect on performance.** **Determine whether the choice of threads of control and their associated responsibilities introduces bottlenecks.**

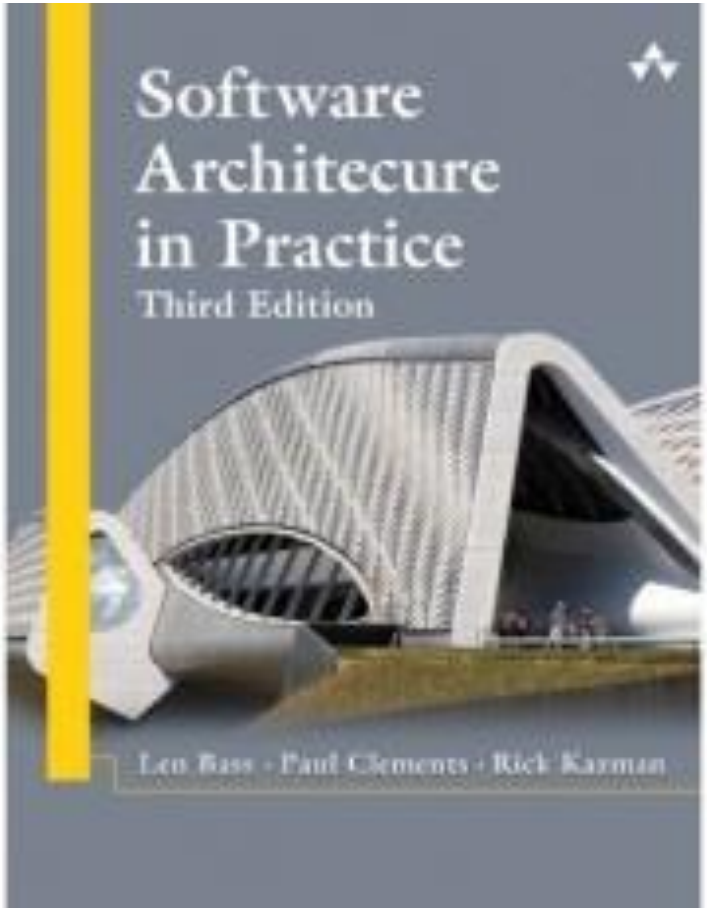
Design Checklist for Performance

Resource Management	**Determine which resources in your system are critical for performance. For these resources ensure they will be monitored and managed under normal and overloaded system operation.** **For example** • **system elements that need to be aware of, and manage, time and other performance-critical resources** • **process/thread models** • **prioritization of resources and access to resources** • **scheduling and locking strategies** • **deploying additional resources on demand to meet increased loads**

Summary

- Performance is about the management of system resources in the face of particular types of demand to achieve acceptable timing behavior.
- Performance can be measured in terms of throughput and latency for both interactive and embedded real time systems.
- Performance can be improved by reducing demand or by managing resources more appropriately.

Chapter 9


Software
Architecure
in Practice
Third Edition
Len Bass · Paul Clements · Rick Kazman

What is Security?

- Security is a measure of the system's ability to protect data and information from unauthorized access while still providing access to people and systems that are authorized.
- An action taken against a computer system with the intention of doing harm is called an *attack* and can take a number of forms.
- It may be an unauthorized attempt to access data or services or to modify data, or it may be intended to deny services to legitimate users.

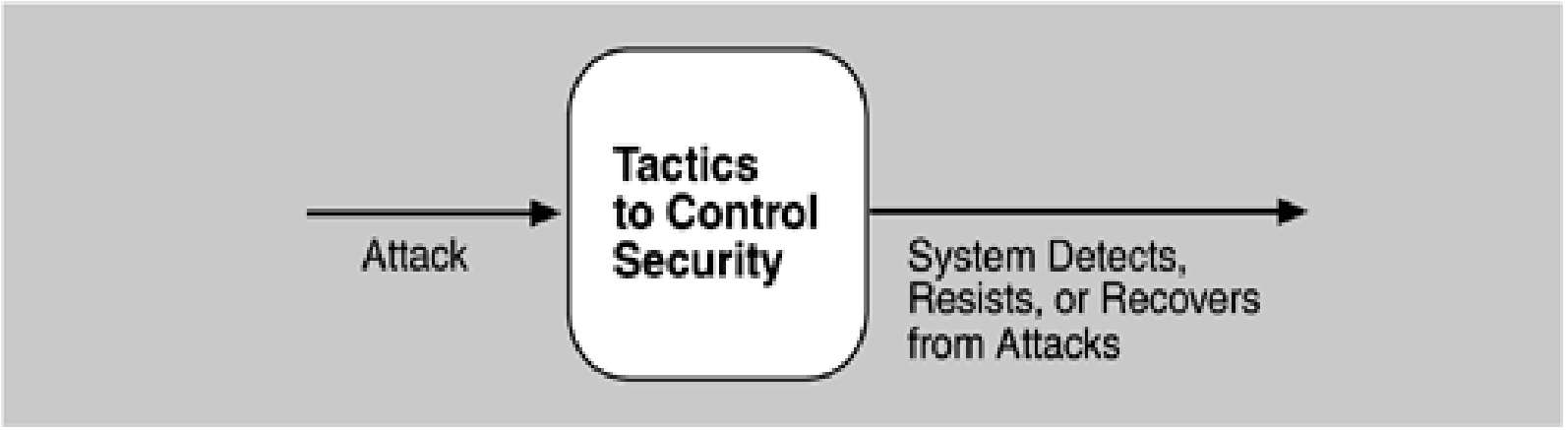
What is Security?

- Security has three main characteristics, called CIA:
  - Confidentiality is the property that data or services are protected from unauthorized access. For example, a hacker cannot access your income tax returns on a government computer.
  - Integrity is the property that data or services are not subject to unauthorized manipulation. For example, your grade has not been changed since your instructor assigned it.
  - Availability is the property that the system will be available for legitimate use. For example, a denial-of-service attack won't prevent you from ordering a book from an online bookstore.
- Other characteristics that support CIA are
  - Authentication verifies the identities of the parties to a transaction and checks if they are truly who they claim to be. For example, when you get an e-mail purporting to come from a bank, authentication guarantees that it actually comes from the bank.
  - Nonrepudiation guarantees that the sender of a message cannot later deny having sent the message and that the recipient cannot deny having received the message. For example, you cannot deny ordering something from the Internet, or the merchant cannot disclaim getting your order.
  - Authorization grants a user the privileges to perform a task. For example, an online banking system authorizes a legitimate user to access his account.

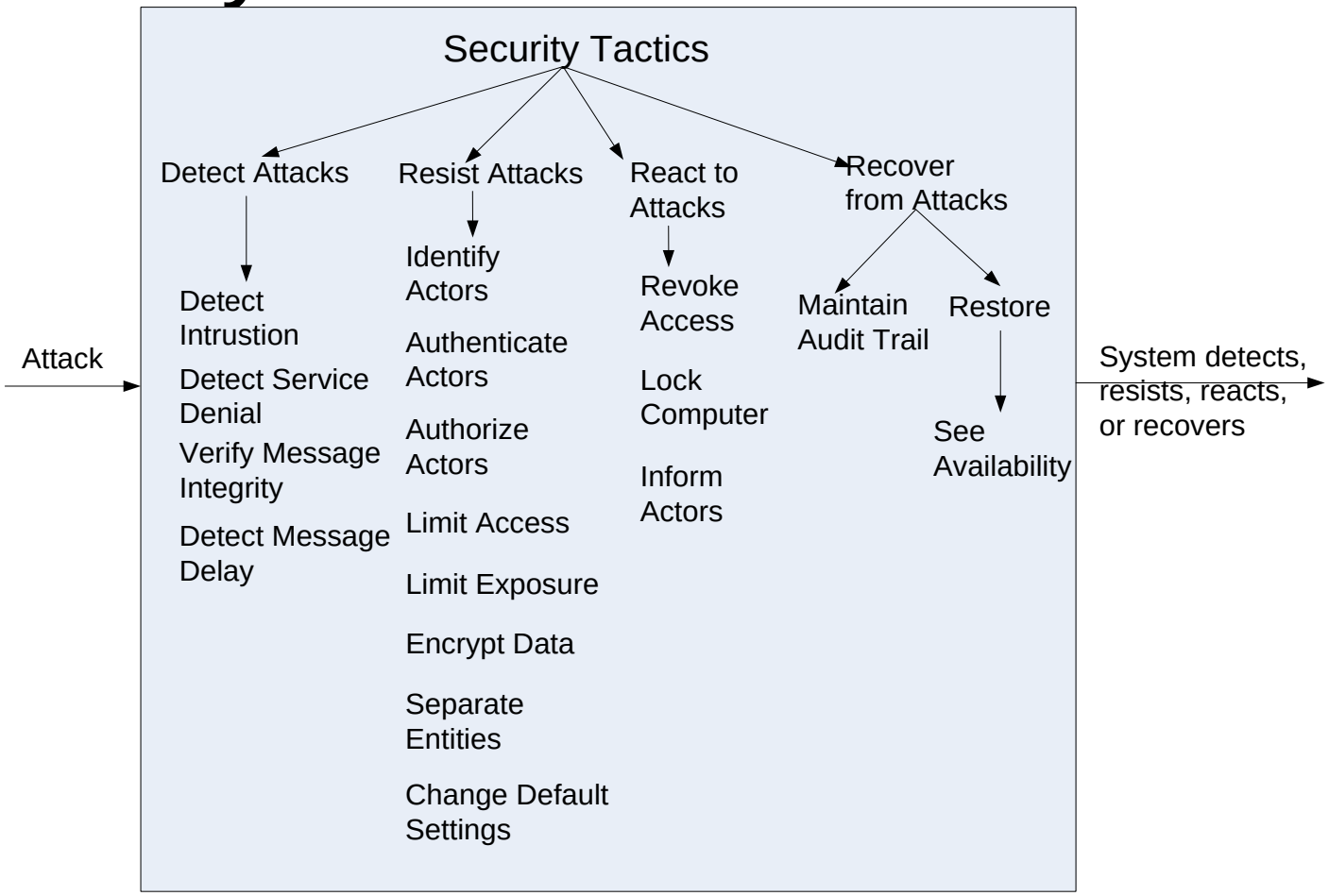
Goal of Security Tactics

- One method for thinking about system security is to think about physical security.
- Secure installations have limited access to them (e.g., by using security checkpoints), have means of detecting intruders (e.g., by requiring legitimate visitors to wear badges), have deterrence mechanisms such as armed guards, have reaction mechanisms such as automatic locking of doors and have recovery mechanisms such as off-site back up.
- This leads to our four categories of tactics: detect, resist, react, and recover.

Goal of Security Tactics



Security Tactics



Detect Attacks

- Detect Intrusion: compare network traffic or service request patterns *within* a system to a set of signatures or known patterns of malicious behavior stored in a database.
- Detect Service Denial: comparison of the pattern or signature of network traffic *coming into* a system to historic profiles of known Denial of Service (DoS) attacks.
- Verify Message Integrity: use techniques such as checksums or hash values to verify the integrity of messages, resource files, deployment files, and configuration files.
- Detect Message Delay: checking the time that it takes to deliver a message, it is possible to detect suspicious timing behavior.

Resist Attacks

- Identify Actors: identify the source of any external input to the system.
- Authenticate Actors: ensure that an actor (user or a remote computer) is actually who or what it purports to be.
- Authorize Actors: ensuring that an authenticated actor has the rights to access and modify either data or services.
- Limit Access: limiting access to resources such as memory, network connections, or access points.

Resist Attacks

- Limit Exposure: minimize the attack surface of a system by having the fewest possible number of access points.
- Encrypt Data: apply some form of encryption to data and to communication.
- Separate Entities: can be done through physical separation on different servers attached to different networks, the use of virtual machines, or an “air gap”.
- Change Default Settings: Force the user to change settings assigned by default.

React to Attacks

- Revoke Access: limit access to sensitive resources, even for normally legitimate users and uses, if an attack is suspected.
- Lock Computer: limit access to a resource if there are repeated failed attempts to access it.
- Inform Actors: notify operators, other personnel, or cooperating systems when an attack is suspected or detected.

Recover From Attacks

- In addition to the Availability tactics for recovery of failed resources there is Audit.
- Audit: keep a record of user and system actions and their effects, to help trace the actions of, and to identify, an attacker.

Design Checklist for Security

Allocation of Responsibilities	**Determine which system responsibilities need to be secure. For each of these responsibilities ensure that additional responsibilities have been allocated to:** • **identify the actor** • **authenticate the actor** • **authorize actors** • **grant or deny access to data or services** • **record attempts to access or modify data or services** • **encrypt data** • **recognize reduced availability for resources or services and inform appropriate personnel and restrict access** • **recover from an attack** • **verify checksums and hash values**

Design Checklist for Security

Coordination Model	**Determine mechanisms required to communicate and coordinate with other systems or individuals. For these communications, ensure that mechanisms for authenticating and authorizing the actor or system, and encrypting data for transmission across the connection are in place.** **Ensure also that mechanisms exist for monitoring and recognizing unexpectedly high demands for resources or services as well as mechanisms for restricting or terminating the connection.**

Design Checklist for Security

Data Model	**Determine the sensitivity of different data fields. For each data abstraction** • **Ensure that data of different sensitivity is separated.** • **Ensure that data of different sensitivity has different access rights and that access rights are checked prior to access.** • **Ensure that access to sensitive data is logged and that the log file is suitably protected.** • **Ensure that data is suitably encrypted and that keys are separated from the encrypted data.** • **Ensure that data can be restored if it is inappropriately modified.**

Design Checklist for Security

Mapping Among Architectural Elements	**Determine how alternative mappings of architectural elements may change how an individual or system may read, write, or modify data, access system services or resources, or reduce their availability. Determine how alternative mappings may affect the recording of access to data, services or resources and the recognition of high demands for resources.** **For each such mapping, ensure that there are responsibilities to** • **identify an actor** • **authenticate an actor** • **authorize actors** • **grant or deny access to data or services** • **record attempts to access or modify data or services** • **encrypt data** • **recognize reduced availability for resources or services, inform appropriate personnel, and restrict access** • **recover from an attack**

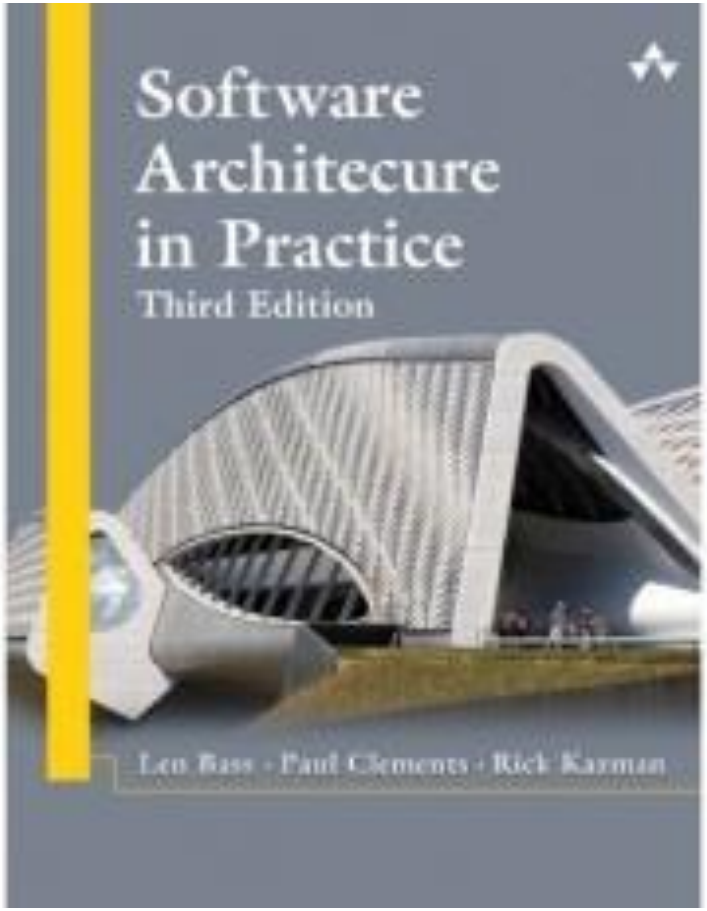
Design Checklist for Security

Resource Management	**Determine the system resources required to identify and monitor a system or an individual who is internal or external, authorized or not authorized, with access to specific resources or all resources.** **Determine the resources required to authenticate the actor, grant or deny access to data or resources, notify appropriate entities, record attempts to access data or resources, encrypt data, recognize high demand for resources, inform users or systems, and restrict access.** **For these resources consider whether an external entity can access or exhaust a critical resource; how to monitor the resource; how to manage resource utilization; how to log resource utilization and ensure that there are sufficient resources to perform necessary security operations.** **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.**

Summary

- Attacks against a system can be characterized as attacks against the confidentiality, integrity, or availability of a system or its data.
- This leads to many of the tactics used to achieve security. Identifying, authenticating, and authorizing actors are tactics intended to determine which users or systems are entitled to what kind of access to a system.
- No security tactic is foolproof and systems *will* be compromised. Hence, tactics exist to detect an attack, limit the spread of any attack, and to react and recover from an attack.

Chapter 10


Software
Architecure
in Practice
Third Edition
Len Bass · Paul Clements · Rick Kazman

What is Testability?

- Software testability refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing.
- Specifically, testability refers to the probability, assuming that the software has at least one fault, that it will fail on its *next* test execution.
- If a fault is present in a system, then we want it to fail during testing as quickly as possible.

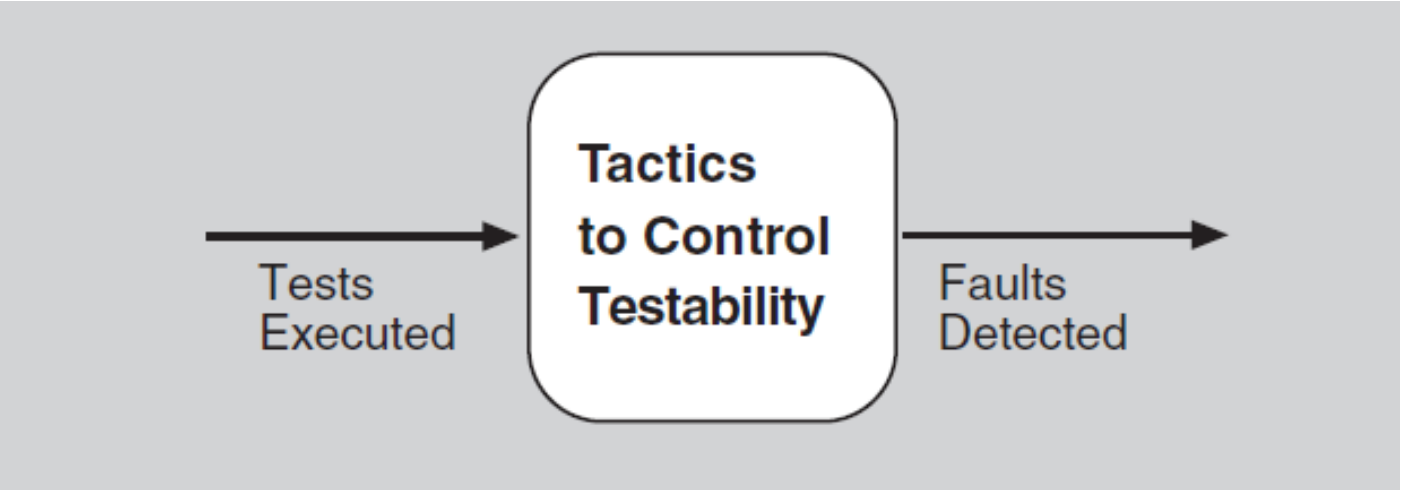
What is Testability?

- For a system to be properly testable, it must be possible to *control* each component's inputs (and possibly manipulate its internal state) and then to *observe* its outputs (and possibly its internal state).

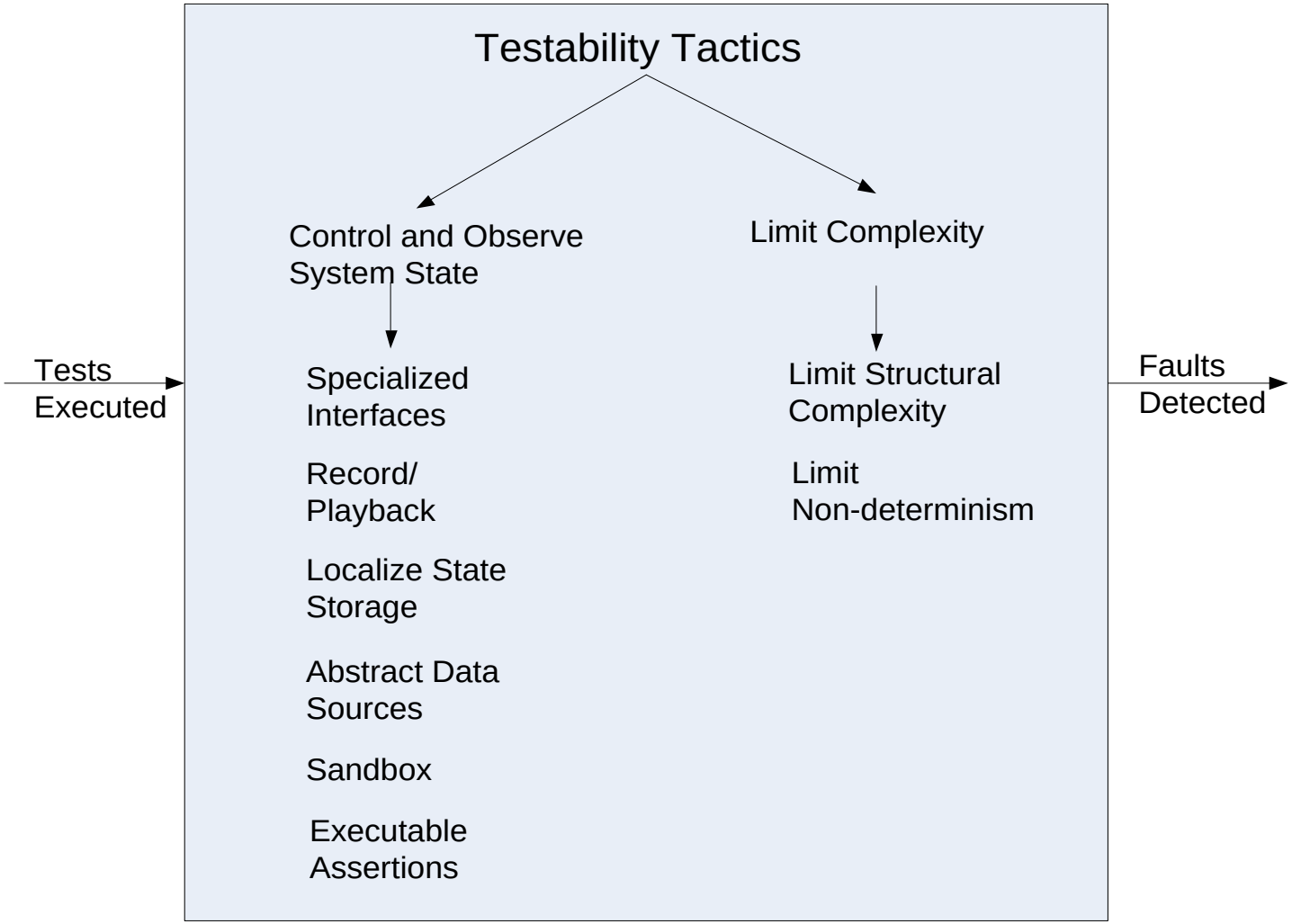
Goal of Testability Tactics

- The goal of tactics for testability is to allow for easier testing when an increment of software development has completed.
- Anything the architect can do to reduce the high cost of testing will yield a significant benefit.
- There are two categories of tactics for testability:
  - The first category deals with adding controllability and observability to the system.
  - The second deals with limiting complexity in the system's design.

Goal of Testability Tactics



Testability Tactics



Control and Observe System State

- Specialized Interfaces: to control or capture variable values for a component either through a test harness or through normal execution.
- Record/Playback: capturing information crossing an interface and using it as input for further testing.
- Localize State Storage: To start a system, subsystem, or module in an arbitrary state for a test, it is most convenient if that state is stored in a single place.

Control and Observe System State

- Abstract Data Sources: Abstracting the interfaces lets you substitute test data more easily.
- 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.

Limit Complexity

- Limit Structural Complexity: avoiding or resolving cyclic dependencies between components, isolating and encapsulating dependencies on the external environment, and reducing dependencies between components in general.
- Limit Non-determinism: finding all the sources of non-determinism, such as unconstrained parallelism, and weeding them out as far as possible.

Design Checklist for Testability

Allocation of Responsibilities	**Determine which system responsibilities are most critical and hence need to be most thoroughly tested.** **Ensure that additional system responsibilities have been allocated to do the following:** • **execute test suite and capture results (external test or self-test)** • **capture (log) the activity that resulted in a fault or that resulted in unexpected (perhaps emergent) behavior that was not necessarily a fault** • **control and observe relevant system state for testing** **Make sure the allocation of functionality provides high cohesion, low coupling, strong separation of concerns, and low structural complexity.**

Design Checklist for Testability

Coordination Model	**Ensure the system's coordination and communication mechanisms:** . **support the execution of a test suite and capture of the results within a system or between systems** . **support capturing activity that resulted in a fault within a system or between systems** . **support injection and monitoring of state into the communication channels for use in testing, within a system or between systems** . **do not introduce needless non-determinism**

Design Checklist for Testability

Data Model	**Determine the major data abstractions that must be tested to ensure the correct operation of the system.** • **Ensure that it is possible to capture the values of instances of these data abstractions.** • **Ensure that the values of instances of these data abstractions can be set when state is injected into the system, so that system state leading to a fault may be re-created.** • **Ensure that the creation, initialization, persistence, manipulation, translation, and destruction of instances of these data abstractions can be exercised and captured**

Design Checklist for Testability

Mapping Among Architectural Elements	**Determine how to test the possible mappings of architectural elements (especially mappings of processes to processors, threads to processes, modules to components) so that the desired test response is achieved and potential race conditions identified.** **In addition, determine whether it is possible to test for illegal mappings of architectural elements.**

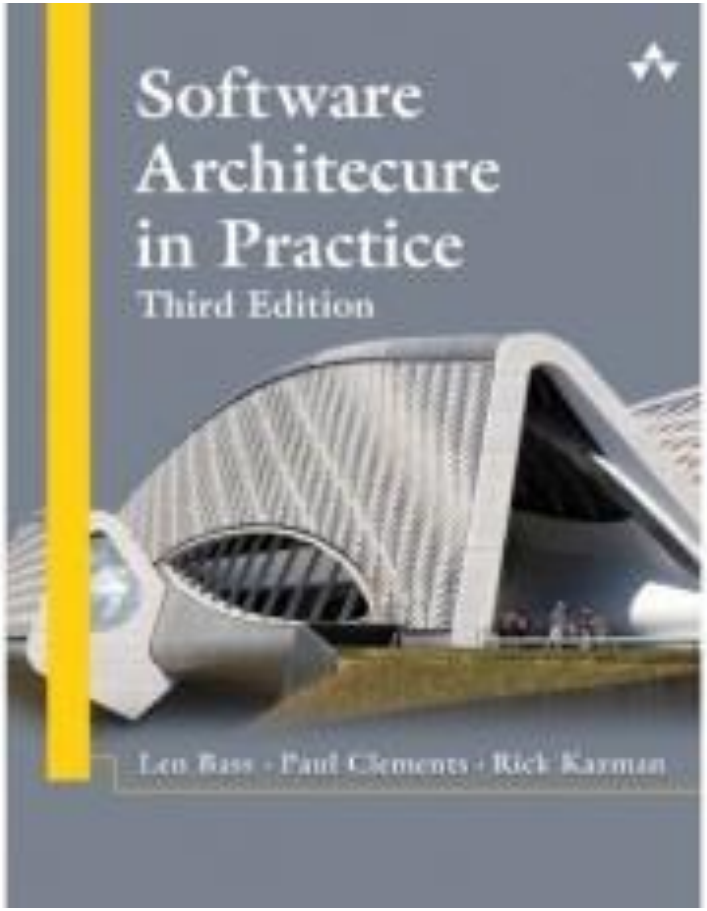
Design Checklist for Testability

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 (or better yet, identical to) 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**

Summary

- Ensuring that a system is easily testable has payoffs both in terms of the cost of testing and the reliability of the system.
- Controlling and observing the system state are a major class of testability tactics.
- Complex systems are difficult to test because of the large state space in which their computations take place, and because of the larger number of interconnections among the elements of the system. Consequently, keeping the system simple is another class of tactics that supports testability.

Chapter 11


Software
Architecure
in Practice
Third Edition
Len Bass · Paul Clements · Rick Kazman

What is Usability?

- Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides.
- Over the years, a focus on usability has shown itself to be one of the cheapest and easiest ways to improve a system's quality (or, more precisely, the user's *perception* of quality).

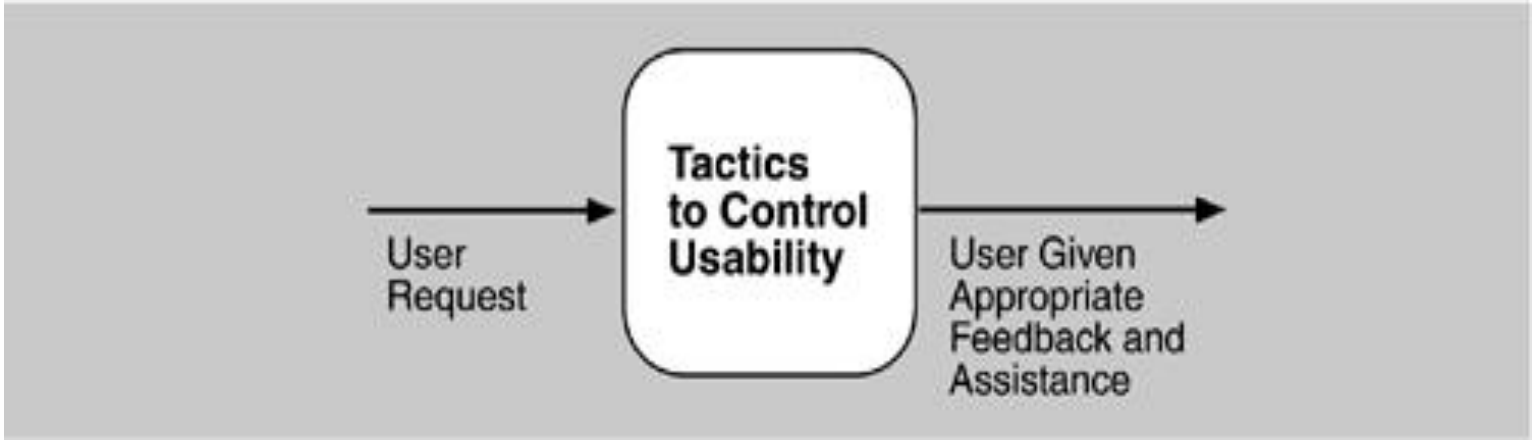
What is Usability?

- Usability comprises the following areas:
  - Learning system features.
  - Using a system efficiently.
  - Minimizing the impact of errors.
  - Adapting the system to user needs.
  - Increasing confidence and satisfaction.

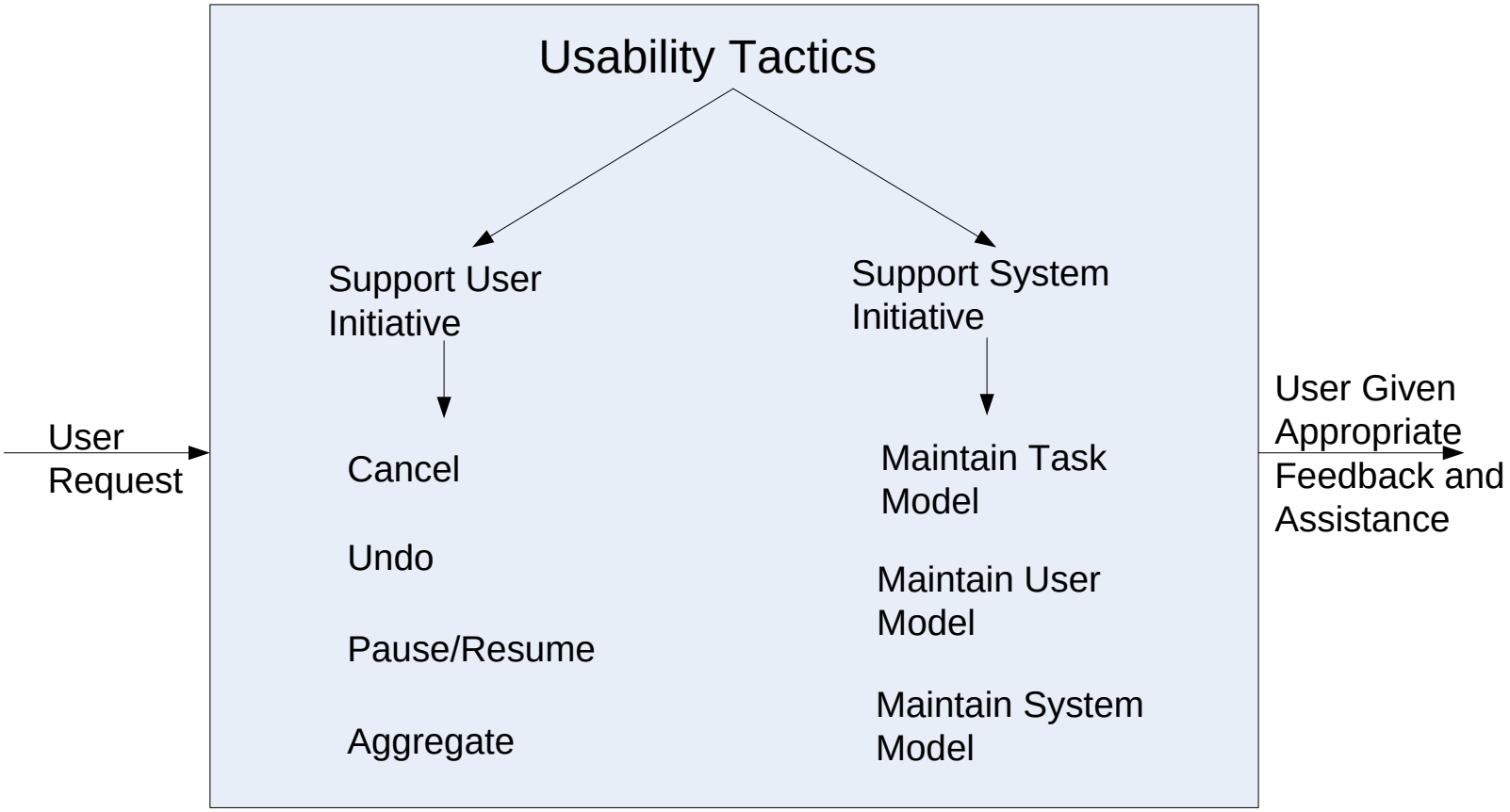
Goal of Usability Tactics

- Researchers in human-computer interaction have used the terms "user initiative," "system initiative," and "mixed initiative" to describe which of the human-computer pair takes the initiative in performing certain actions and how the interaction proceeds.
- Usability scenarios can combine initiatives from both perspectives.
- We use this distinction between user and system initiative to discuss the tactics that the architect uses to achieve the various scenarios.

Goal of Usability Tactics



Usability Tactics



Support User Initiative

- Cancel: the system must listen for the cancel request; the command being canceled must be terminated; resources used must be freed; and collaborating components must be informed.
- Pause/Resume: temporarily free resources so that they may be re-allocated to other tasks.
- Undo: maintain a sufficient amount of information about system state so that an earlier state may be restored, at the user's request.
- Aggregate: ability to aggregate lower-level objects into a group, so that a user operation may be applied to the group, freeing the user from the drudgery.

Support System Initiative

- Maintain Task Model: determines context so the system can have some idea of what the user is attempting and provide assistance.
- Maintain User Model: explicitly represents the user's knowledge of the system, the user's behavior in terms of expected response time, etc.
- Maintain System Model: system maintains an explicit model of itself. This is used to determine expected system behavior so that appropriate feedback can be given to the user.

Design Checklist for Usability

Allocation of Responsibilities	**Ensure that additional system responsibilities have been allocated, as needed, to assist the user in** . **learning how to use the system** . **efficiently achieving the task at hand** . **adapting and configuring the system** . **recovering from user and system errors**

Design Checklist for Usability

Coordination Model	**Determine whether the properties of system elements' coordination—timeliness, currency, completeness, correctness, consistency—affect how a user learns to use the system, achieves goals or completes tasks, adapts and configures the system, recovers from user and system errors, increases confidence and satisfaction.** **For example, can the system respond to mouse events and give semantic feedback in real time? Can long-running events be canceled in a reasonable amount of time?**

Design Checklist for Usability

Data Model	**Determine the major data abstractions that are involved with user-perceivable behavior.** **Ensure these major data abstractions, their operations, and their properties have been designed to assist the user in achieving the task at hand, adapting and configuring the system, recovering from user and system errors, learning how to use the system, and increasing satisfaction and user confidence** **For example, the data abstractions should be designed to support undo and cancel operations: the transaction granularity should not be so great that canceling or undoing an operation takes an excessively long time.**

Design Checklist for Usability

Mapping Among Architectural Elements	**Determine what mapping among architectural elements is visible to the end user (for example, the extent to which the end user is aware of which services are local and which are remote).** **For those that are visible, determine how this affects the ways in which, or the ease with which the user will learn how to use the system, achieve the task at hand, adapt and configure the system, recover from user and system errors, and increase confidence and satisfaction.**

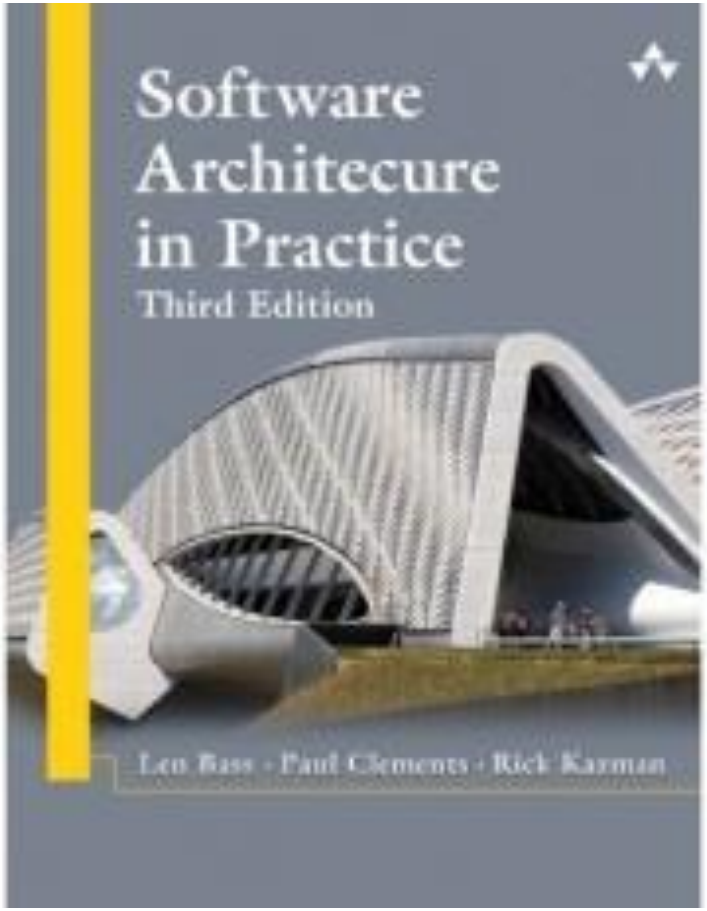
Design Checklist for Usability

Resource Management	**Determine how the user can adapt and configure the system's use of resources.** **Ensure that resource limitations under all user-controlled configurations will not make users less likely to achieve their tasks. For example, attempt to avoid configurations that would result in excessively long response times.** **Ensure that the level of resources will not affect the users' ability to learn how to use the system, or decrease their level of confidence and satisfaction with the system.**

Summary

- Architectural support for usability involves both allowing the user to take the initiative in circumstances such as cancelling a long running command, undoing a completed command, and aggregating data and commands.
- To predict user or system response, the system must keep a model of the user, the system, and the task.

Chapter 12


Software
Architecure
in Practice
Third Edition
Len Bass · Paul Clements · Rick Kazman

Other Important Quality Attributes

- Variability: is a special form of modifiability. It refers to the ability of a system and its supporting artifacts to support the production of a set of variants that differ from each other in a preplanned fashion.
- Portability: is also a special form of modifiability. Portability refers to the ease with which software that built to run on one platform can be changed to run on a different platform.
- Development Distributability: is the quality of designing the software to support distributed software development.

Other Important Quality Attributes

- Scalability: Horizontal scalability (scaling out) refers to adding more resources to logical units such as adding another server to a cluster. Vertical scalability (scaling up) refers to adding more resources to a physical unit such as adding more memory to a computer.
- Deployability: is concerned with how an executable arrives at a host platform and how it is invoked.
- Mobility: deals with the problems of movement and affordances of a platform (e.g. size, type of display, type of input devices, availability and volume of bandwidth, and battery life).

Other Important Quality Attributes

- Monitorability: deals with the ability of the operations staff to monitor the system while it is executing.
- Safety: Software safety is about the software's ability to avoid entering states that cause or lead to damage, injury, or loss of life, and to recover and limit the damage when it does enter into bad states. The architectural concerns with safety are almost identical with those for availability (i.e. preventing, detecting, and recovering from failures).

Other Categories of Quality Attributes

- Conceptual Integrity: refers to consistency in the design of the architecture. It contributes to the understandability of the architecture. Conceptual integrity demands that the same thing is done in the same way through the architecture.
- Marketability: Some systems are marketed by their architectures, and these architectures sometimes carry a meaning all their own, independent of what other quality attributes they bring to the system (e.g. service-oriented or cloud-based).

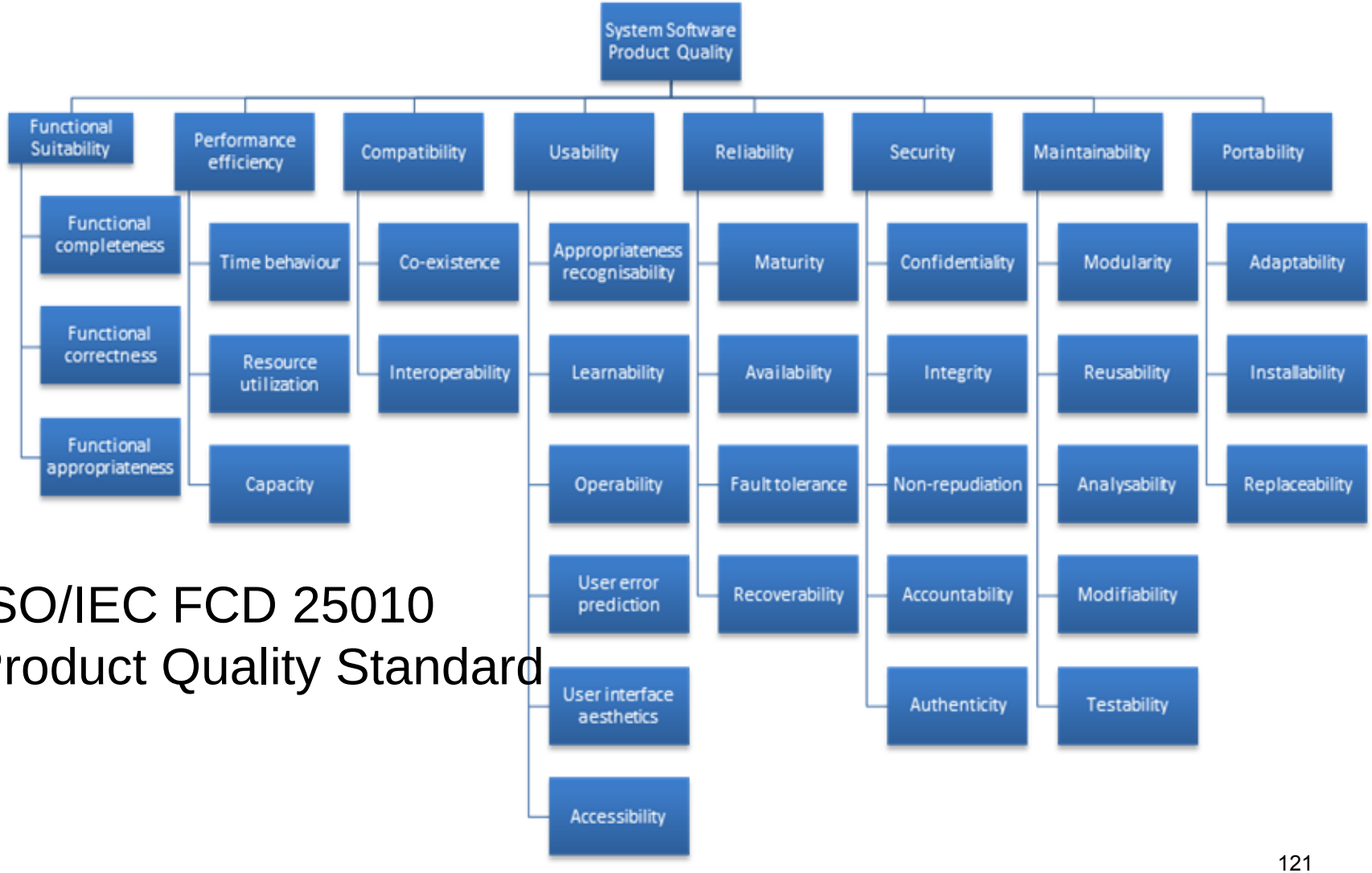
Other Categories of Quality Attributes

- Quality in Use: qualities that pertain to the use of the system by various stakeholders. For example
  - Effectiveness: a measure whether the system is correct
  - Efficiency: the effort and time required to develop a system
  - Freedom from risk: degree to which a product or system affects economic status, human life, health, or the environment

Software Quality Attributes and System Quality Attributes

- Physical systems, such as aircraft or automobiles or kitchen appliances, that rely on software embedded within are designed to meet a whole other litany of quality attributes: weight, size, electric consumption, power output, pollution output, weather resistance, battery life, and on and on.
- The software architecture can have a substantial effect on the system's quality attributes.

Standard Lists of Quality Attributes



ISO/IEC FCD 25010
Product Quality Standard

Standard Lists of Quality Attributes

- Advantages:
  - Can be helpful checklists to assist requirements gatherers in making sure that no important needs were overlooked.
  - Can serve as the basis for creating your own checklist that contains the quality attributes of concern in your domain, your industry, your organization, your products, …

Standard Lists of Quality Attributes

- Disadvantages:
  - No list will ever be complete.
  - Lists often generate more controversy than understanding.
  - Lists often purport to be *taxonomies*. But what is a denial-of-service attack?
  - They force architects to pay attention to every quality attribute on the list, even if only to finally decide that the particular quality attribute is irrelevant to their system.

Dealing with “X-ability”

- Suppose you must deal with a quality attribute for which there is no compact body of knowledge, e.g. green computing.
- What do you do?
  1. Model the quality attribute
  2. Assemble a set of tactics for the quality attribute
  3. Construct design checklists

Summary

- There are many other quality attributes than the seven that we cover in detail.
- Taxonomies of attributes may offer some help, but their disadvantages often outweigh their advantages.
- You may need to design or analyze a system for a “new” quality attribute. While this may be challenging, it is doable.