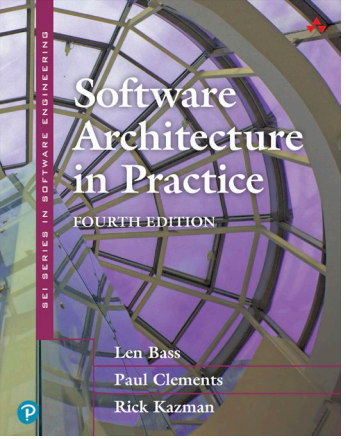
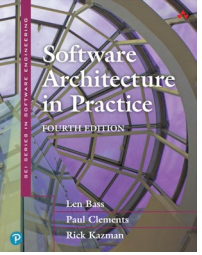


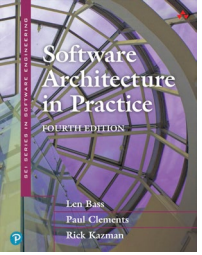
Chapter 4: Availability

Technology does not always rhyme with perfection and reliability. Far from it in reality!
—Jean-Michel Jarre



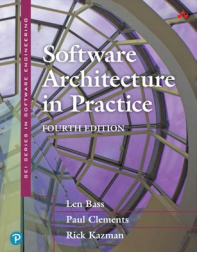
Chapter Outline

- What is Availability?
- Availability General Scenario
- Tactics for Availability
- Tactics-Based Questionnaire for Availability
- Patterns for Availability
- Summary



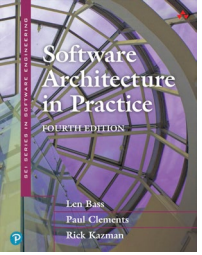
What is Availability?

- Availability refers to a property of software that it is there and ready to carry out its task when you need it to be.
- Availability encompasses the ability of a system to mask or repair *faults* such that they do not become *failures*.
- Availability builds on *reliability* by adding the notion of recovery (repair). The goal is to minimize service outage time by mitigating faults.
- The time to repair is the time until the failure is no longer observable.



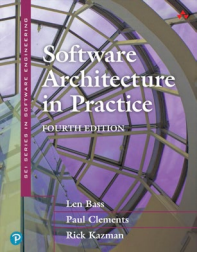
What is Availability?

- A failure's cause is a *fault*. A fault can be internal or external to the system.
- Faults can be:
  - prevented,
  - tolerated,
  - removed,
  - forecast.
- Through these actions, a system becomes "resilient" to faults.



What is Availability?

- We are concerned with:
  - how faults are detected,
  - how frequently they occur,
  - what happens when they occur,
  - how long a system may be out of operation,
  - how faults or failures can be prevented, and
  - what notifications are required when a failure occurs.

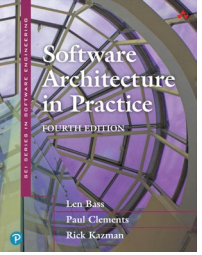


What is Availability?

- We often measure availability properties such as:
  - *MTBF:* the mean time between failures
  - *MTTR:* the mean time to repair
- Steady-state availability is calculated as:

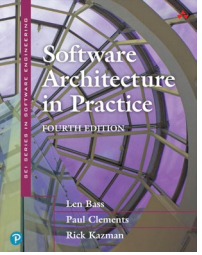
$$\boldsymbol{MTBF}/(\boldsymbol{MTBF} + \boldsymbol{MTTR})$$

- This is how we calculate measures such as "99.99% availability" (often seen in SLAs).



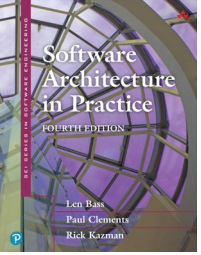
Example Availability Measures

Availability	Downtime/90 Days	Downtime/Year
99.0%	21 hr, 36 min	3 days, 15.6 hr
99.9%	2 hr, 10 min	8 hr, 0 min, 46 sec
99.99%	12 min, 58 sec	52 min, 34 sec
99.999%	1 min, 18 sec	5 min, 15 sec
99.9999%	8 sec	32 sec



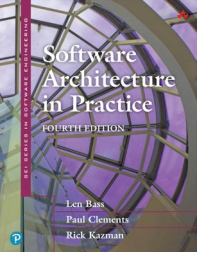
Availability General Scenario

Portion of Scenario	Description	Possible Values
Source	This specifies where the fault comes from.	Internal/external: people, hardware, software, physical infrastructure, physical environment
Stimulus	The stimulus to an availability scenario is a fault.	Fault: omission, crash, incorrect timing, incorrect response
Artifact	This specifies which portions of the system are responsible for and affected by the fault.	Processors, communication channels, storage, processes, affected artifacts in the system's environment
Environment	We may be interested in not only how a system behaves in its "normal" environment, but also how it behaves in situations such as when it is already recovering from a fault.	Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation



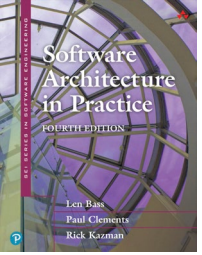
Availability General Scenario

Response	The most commonly desired response is to prevent the fault from becoming a failure, but other responses may also be important, such as notifying people or logging the fault for later analysis. This section specifies the desired system response.	Prevent the fault from becoming a failure Detect the fault: ▪ Log the fault ▪ Notify the appropriate entities (people or systems) ▪ Recover from the fault ▪ Disable the source of events causing the fault ▪ Be temporarily unavailable while a repair is being effected ▪ Fix or mask the fault/failure or contain the damage it causes ▪ Operate in a degraded mode while a repair is being effected
Response measure	We may focus on a number of measures of availability, depending on the criticality of the service being provided.	▪ Time or time interval when the system must be available ▪ Availability percentage (e.g., 99.999 percent) ▪ 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 percent) or rate (e.g., up to 100 per second) of a certain class of faults that the system prevents, or handles without failing

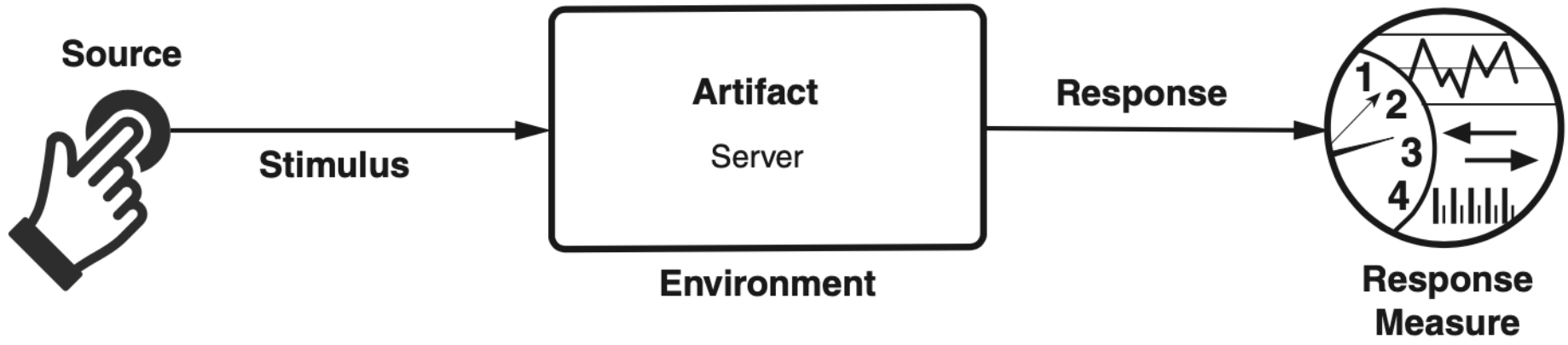


Sample Concrete Availability Scenario

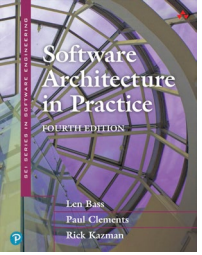
- *A server in a server farm fails during normal operation, and the system informs the operator and continues to operate with no downtime*.



Sample Concrete Availability Scenario

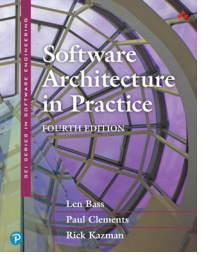


Source	Stimulus	Environment	Response	Response Measure
Server in a server farm	Server fails	Normal operation	System informs operator System continues to operate	No downtime

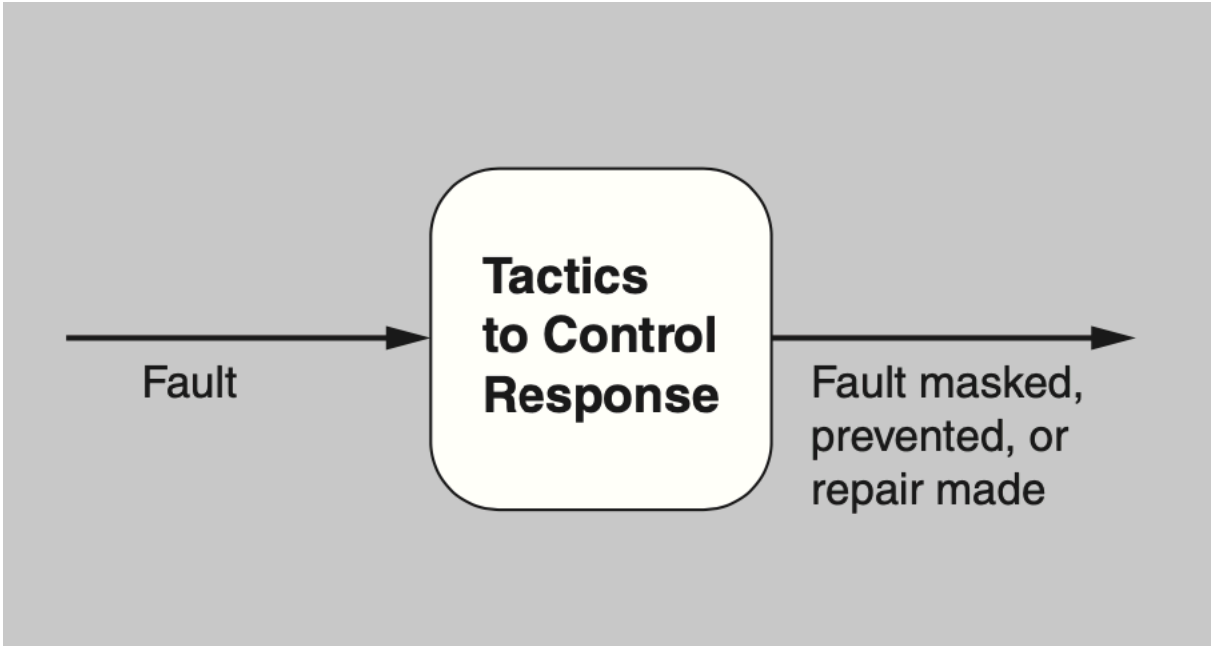


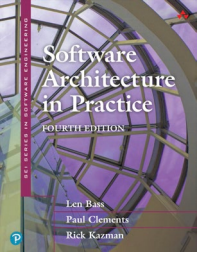
Goal of Availability Tactics

- 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.

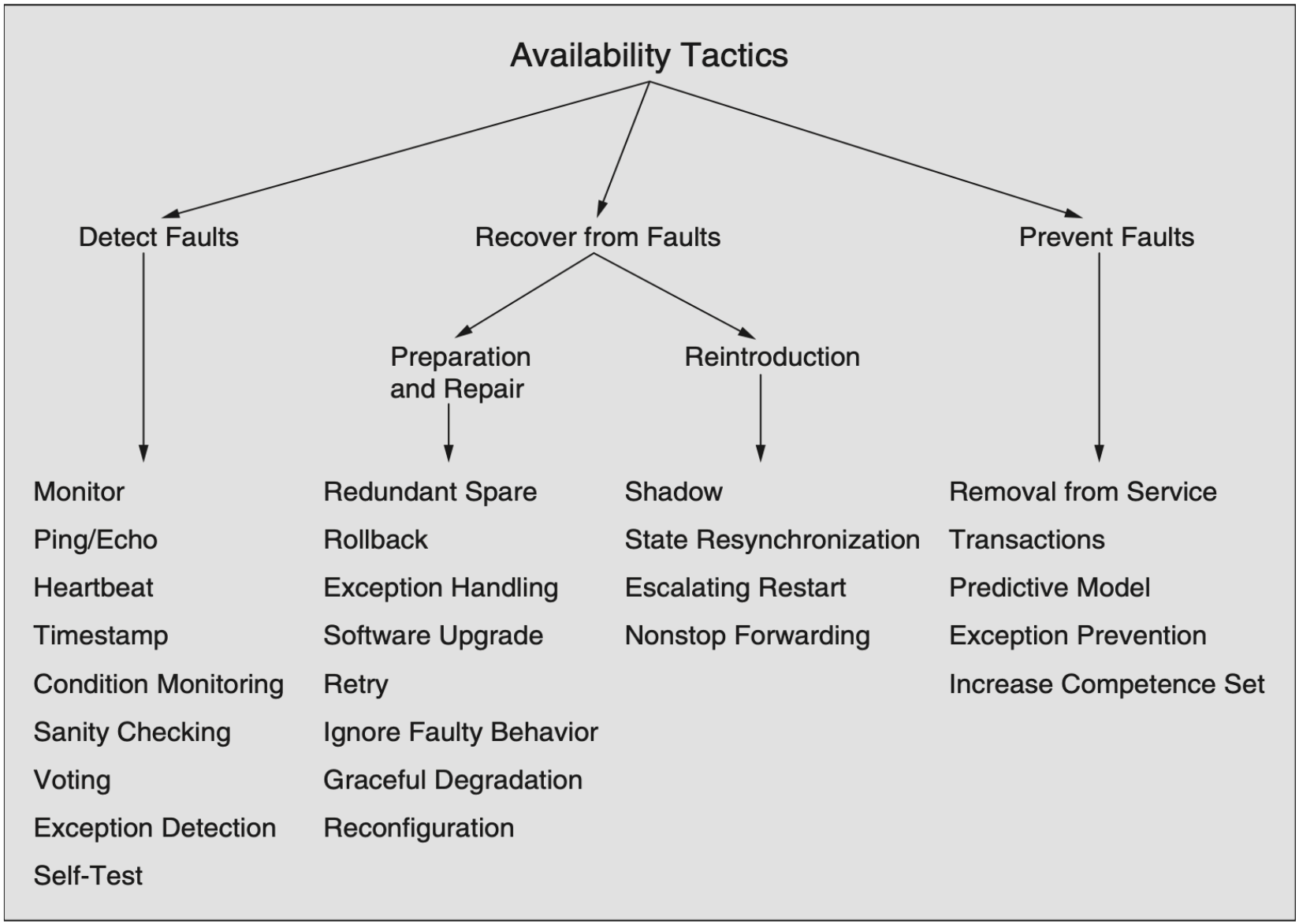


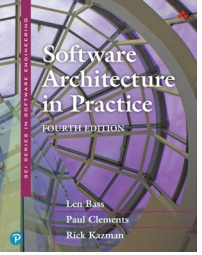
Goal of Availability Tactics





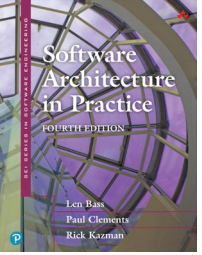
Availability Tactics





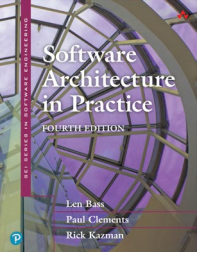
Detect Faults

- 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.



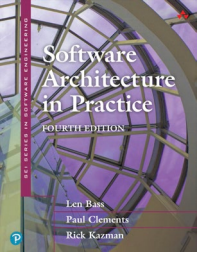
Detect Faults

- 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.



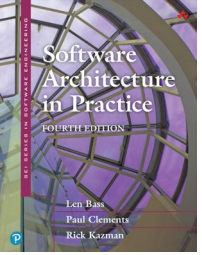
Detect Faults

- 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.



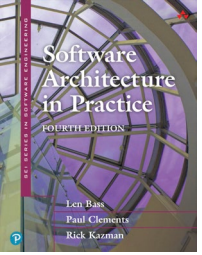
Recover from Faults (Preparation & Repair)

- *Redundant spare.* This tactic refers to a configuration in which one or more duplicate components can step in and take over the work if the primary component fails.
  - This tactic is at the heart of the hot spare, warm spare, and cold spare patterns, which differ primarily in how up-to-date the backup component is at the time of its takeover.



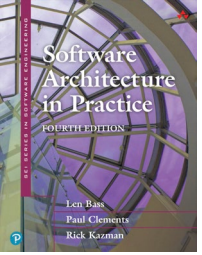
Recover from Faults (Preparation & Repair)

- 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.



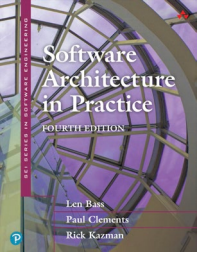
Recover from Faults (Preparation & Repair)

- 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.
- Graceful 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.



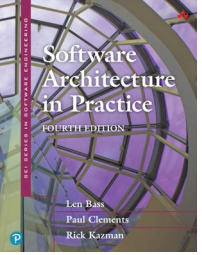
Recover from Faults (Reintroduction)

- 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.



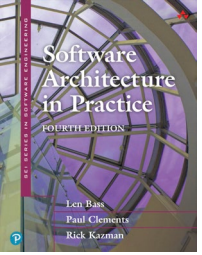
Prevent Faults

- 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.



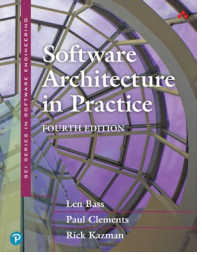
Prevent Faults

- 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.



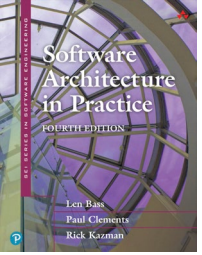
Tactics-Based Questionnaire for Availability

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Detect Faults	Does the system use **ping/ echo** to detect failure of a component or connection, or network congestion?				
	Does the system use a component 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.				
	Does the system use a **heartbeat**—a periodic message exchange between a system monitor and a process—to detect failure of a component or connection, or network congestion?				
	Does the system use a **timestamp** to detect incorrect sequences of events in distributed systems?				
	Does the system use **voting** to check that replicated components are producing the same results?				
	The replicated components may be identical replicas, functionally redundant, or analytically redundant.				
	Does the system use **exception detection** to detect a system condition that alters the normal flow of execution (e.g., system exception, parameter fence, parameter typing, timeout)?				
	Can the system do a **self-test** to test itself for correct operation?				



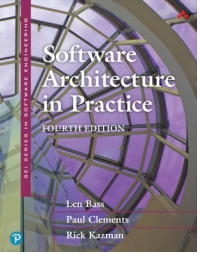
Tactics-Based Questionnaire for Availability

Recover from Faults (Preparation and Repair)	Does the system employ **redundant spares**?
	Is a component's role as active versus spare fixed, or does it change in the presence of a fault? What is the switchover mechanism? What is the trigger for a switchover? How long does it take for a spare to assume its duties?
	Does the system employ **exception handling** to deal with faults?
	Typically the handling involves either reporting, correcting, or masking the fault.
	Does the system employ **rollback**, so that it can revert to a previously saved good state (the "rollback line") in the event of a fault?
	Can the system perform in-service **software upgrades** to executable code images in a non-service-affecting manner?
	Does the system systematically **retry** in cases where the component or connection failure may be transient?
	Can the system simply ignore faulty behavior (e.g., ignore messages when it is determined that those messages are spurious)?
	Does the system have a policy of degradation when resources are compromised, maintaining the most critical system functions in the presence of component failures, and dropping less critical functions?
	Does the system have consistent policies and mechanisms for **reconfiguration** after failures, reassigning responsibilities to the resources left functioning, while maintaining as much functionality as possible?



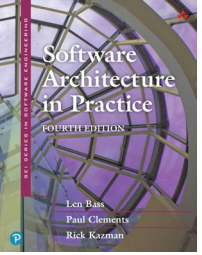
Tactics-Based Questionnaire for Availability

Recover from Faults (Reintroduction)	Can the system operate 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?
	If the system uses active or passive redundancy, does it also employ **state resynchronization** to send state information from active components to standby components?
	Does the system employ **escalating restart** to recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected?
	Can message processing and routing portions of the system employ **nonstop forwarding**, where functionality is split into supervisory and data planes?
Prevent Faults	Can the system **remove components from service**, temporarily placing a system component in an out-of-service state for the purpose of preempting potential system failures?
	Does the system employ **transactions**—bundling state updates so that asynchronous messages exchanged between distributed components are *atomic*, *consistent*, *isolated*, and *durable*?
	Does the system use a **predictive model** to monitor the state of health of a component to ensure that the system is operating within nominal parameters?
	When conditions are detected that are predictive of likely future faults, the model initiates corrective action.



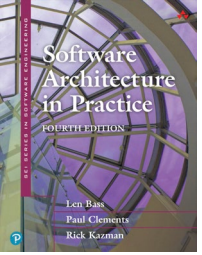
Patterns for Availability

- *Active redundancy (hot spare)*. This refers to a configuration in which all of the nodes in a protection group receive and process identical inputs in parallel, allowing the redundant spare(s) to maintain a synchronous state with the active node(s).
- Because the redundant spare possesses an identical state to the active processor, it can take over from a failed component in a matter of milliseconds.



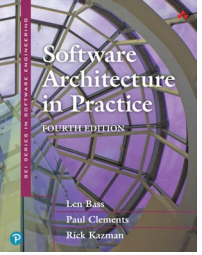
Redundant Spare Patterns for Availability

- *Passive redundancy (warm spare)*. Here 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.
- Because this state is loosely coupled with the active node(s), the redundant nodes are referred to as warm spares.
- Passive redundancy achieves a balance between the more highly available but more compute-intensive (and expensive) active redundancy pattern and the less available but significantly less complex (and cheaper) cold spare pattern.



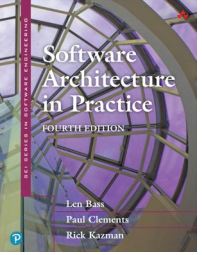
Redundant Spare Patterns for Availability

- *Spare (cold spare)*. Cold sparing refers to a configuration in which redundant spares remain out of service until a failover occurs, at which point a power-on-reset procedure is initiated on the redundant spare prior to its being placed in service.
- Due to its poor recovery performance, and hence its high mean time to repair, this pattern is poorly suited to systems having high-availability requirements



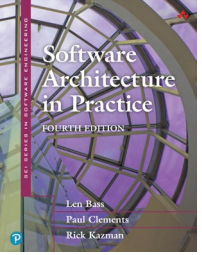
Benefits of Redundant Spare Patterns

- The benefit of a redundant spare is a system that continues to function correctly after only a brief delay in the presence of a failure.
- The alternative is a system that stops functioning correctly (or altogether) until the failed component is repaired.
- This could take hours or days.



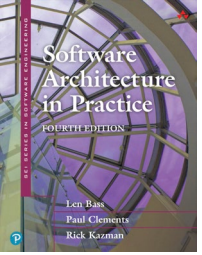
Tradeoffs in Redundant Spare Patterns

- The tradeoff with any of these patterns is the additional cost and complexity incurred in providing a spare.
- The tradeoff among the three alternatives is the time to recover from a failure versus the runtime cost incurred to keep a spare up-to-date.
- A hot spare carries the highest cost but leads to the fastest recovery time, for example.



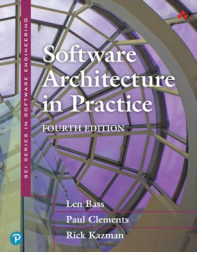
TMR Pattern for Availability

- This widely used implementation of the voting tactic employs three components that do the same thing. Each component receives identical inputs and forwards its output to the voting logic, which detects any inconsistency among the three output states. Faced with an inconsistency, the voter reports a fault.
- It must also decide which output to use, and different instantiations of this pattern use different decision rules. Typical choices are letting the majority rule or choosing some computed average of the disparate outputs.



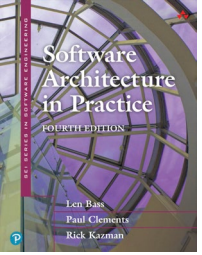
TMR Pattern Benefits/Tradeoffs

- Benefits:
  - TMR is simple to understand and to implement. It is blissfully independent of what might be causing disparate results, and is only concerned about making a reasonable choice so that the system can continue to function.
- Tradeoffs:
  - There is a tradeoff between increasing the level of replication, which raises the cost, and the resulting availability. In systems employing TMR, the statistical likelihood of two or more components failing is vanishingly small, and three components represents a sweet spot between availability and cost.



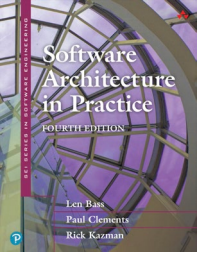
Circuit Breaker Pattern for Availability

- A commonly used availability tactic is retry. In the event of a timeout or fault when invoking a service, the invoker simply tries again—and again, and again. A circuit breaker keeps the invoker from trying countless times, waiting for a response that never comes.
- In this way, it breaks the endless retry cycle when it deems that the system is dealing with a fault. That's the signal for the system to begin handling the fault.
- Until the circuit break is "reset," subsequent invocations will return immediately without passing along the service request.



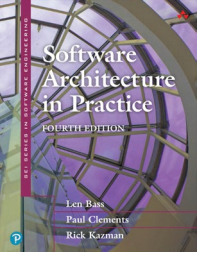
Circuit Breaker Pattern Benefits/Tradeoffs

- Benefits:
  - This pattern can remove from individual components the policy about how many retries to allow before declaring a failure.
  - At worst, endless fruitless retries would make the invoking component as useless as the invoked component that has failed. This problem is especially acute in distributed systems, where you could have many callers calling an unresponsive component and effectively going out of service themselves, causing the failure to cascade across the whole system. The circuit breaker, *in conjunction with software that listens to it and begins recovery procedures*, prevents that problem.
- Tradeoffs:
  - Care must be taken in choosing timeout (or retry) values. If the timeout is too long, then unnecessary latency is added. But if the timeout is too short, then the circuit breaker will be tripping when it does not need to—a kind of "false positive"—which can lower the availability and performance of these services.



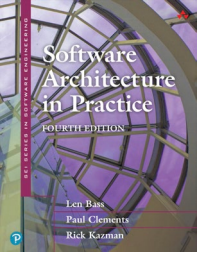
Other Common Availability Patterns

- *Process pairs*. This pattern employs checkpointing and rollback. In case of failure, the backup has been checkpointing and (if necessary) rolling back to a safe state, so is ready to take over when a failure occurs.
- *Forward error recovery*. This pattern provides a way to get out of an undesirable state by *moving forward* to a desirable state. This often relies upon built-in error-correction capabilities, such as data redundancy, so that errors may be corrected without the need to fall back to a previous state or to retry. Forward error recovery finds a safe, possibly degraded, state from which the operation can move forward.



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.