An Introduction to Software Architecture and Design I

Dr. Mark C. Paulk

Mark.Paulk@utdallas.edu, Mark.Paulk@ieee.org

UTDallas, Jonsson School of Engineering and Computer Science

Questions About Architecture

What is “software architecture?”

Is it different from “design?” How?

Why should a software professional care about architecture? Or design?

- **Why should a student care?**

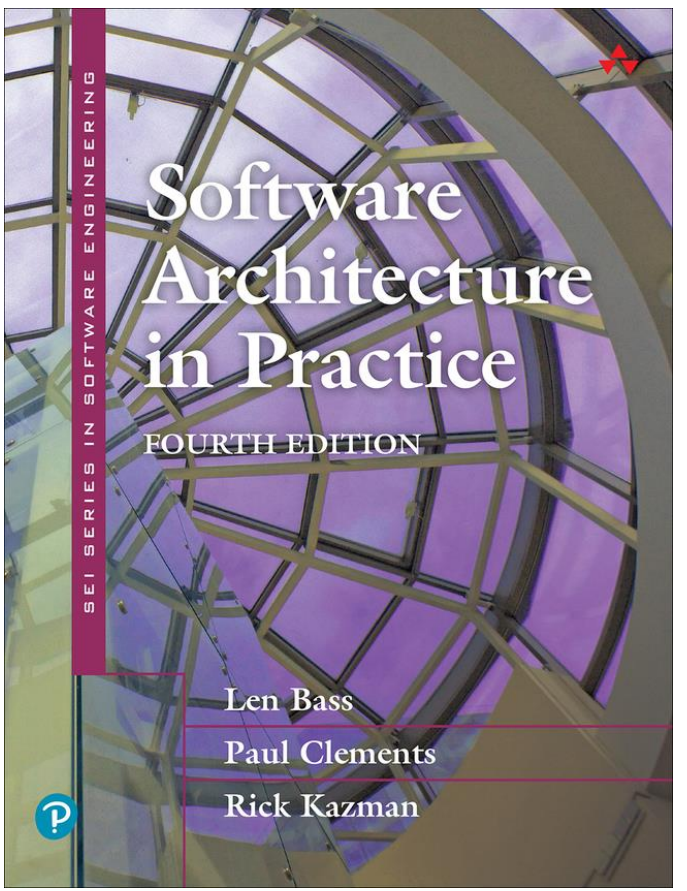
How do we do architectural design?

- **What is involved in an architecture?**

How would we recognize a “good” architecture?

Software Architecture References

L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, Fourth Edition, 2021.



Recommended

- **H. Cervantes and R. Kazman, Designing Software Architectures: A Practical Approach, 2016.**
- **P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, Documenting Software Architectures: Views and Beyond, 2002.**
- **G. Fairbanks, Just Enough Software Architecture, 2010.**
- **A.J. Lattanze, Architecting Software Intensive Systems: A Practitioners Guide, 2008.**
- **R.C. Martin, Clean Architecture, 2017.**

What Is a Software Architecture?

The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both. (Bass 2021)

- **some partition systems into implementation units (modules), which are static**
- **some are dynamic, focusing on the way the elements interact with each other at runtime to carry out the system's functions (component-and-connector)**
- **some describe the mapping from software structures to the system's organizational, developmental, installation, and execution environments (allocation)**

Structures and Views

A view is a representation of a coherent set of architectural elements, as written by and read by system stakeholders.

- **It consists of a representation of a set of elements and the relations among them.**

A structure is the set of elements itself, as they exist in software or hardware.

- **module**
- **communication and coordination (C&C)**
- **allocation**

A view is a representation of a structure.

Early Design Decisions

The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.

- **Will the system run on one processor or be distributed across multiple processors?**
- **Will the software be layered? If so, how many layers will there be? What will each one do?**
- **Will components communicate synchronously or asynchronously? Will they interact by transferring control or data or both?**
- **Will the system depend on specific features of the operating system or hardware?**
- **Will the information that flows through the system be encrypted or not?**
- **What operating system will we use?**
- **What communication protocol will we choose?**

A Good Definition?

Is the Bass, Clements, and Kazman definition of architecture distinguishable from design?

- **at least it focuses on reasoning about the system and its relationships**

Is defining architecture as the early, critical design decisions any better?

- **what is early? what is critical?**

Would characterizing architecture as the top-level design (as opposed to detailed) help?

- **some architecturally significant design decisions are very detailed!**

A Change-Driven Definition

Local change

- **modify a single element**

Non-local change

- **multiple element modifications leaving the underlying architectural approach intact**

Architectural change

- **affects the fundamental ways in which the elements interact with each other**
- **will probably require changes all over the system**

Every System Has an Architecture
(Fairbanks 2010)

Architecture-indifferent design

- **opens the door to complexity…**

Architecture-focused design

Architecture hoisting

- **design the architecture with the intent of guaranteeing a goal or property of the system**
- **you will either find**
  - **code that manages the goal or property**
  - **a deliberate structural constraint (often with reasoning or calculations) that ensures it**

Defining Software Architecture

No single "good" definition of architecture…

Architectural decisions span the entire system or major subsystems.

Architectural decisions need to be made early and are critical because of their span.

Architectural decisions are driven by the system's goals, which must be articulated.

Functionality

Functionality does not determine architecture.

- **Functionality is assigned to specific elements of the system (responsibility driven design).**
- **If functionality were the only thing that mattered, you wouldn't have to divide the system into pieces at all; a single monolithic blob with no internal structure would do just fine.**

The architect's interest in functionality is in how it interacts with and constrains other qualities.

The functional requirements are not (usually) architecturally significant… leaving the non-functional requirements aka quality attributes.

Quality Attribute

A measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders.

You can think of a quality attribute as measuring the "goodness" of a product along some dimension of interest to a stakeholder.

- **A qualification of a functional requirement is an item such as how fast the function must be performed, or how resilient it must be to erroneous input.**
- **A qualification of the overall product is an item such as the time to deploy the product or a limitation on operational costs.**

Common Quality Attributes

Availability
Deployability
Energy Efficiency
Integrability
Modifiability
Performance
Safety
Security
Testability
Usability

Other possible quality attributes include portability, scalability, mobility, … see IEEE/ISO/IEC 25010

Tactics

A tactic is a design decision that influences the achievement of a quality attribute response.

The focus of a tactic is on a single quality attribute response.

- **Within a tactic, there is no consideration of tradeoffs.**
- **Tradeoffs must be explicitly considered and controlled by the designer.**
- **In this respect, tactics differ from architectural patterns, where tradeoffs are built into the pattern.**

Specifying Quality Attribute Requirements

Source of stimulus

- **some entity (a human, a computer system, or any other actuator) that generated the stimulus**

Stimulus

- **a condition that requires a response when it arrives at a system**

Environment

- **the stimulus occurs under certain conditions**

Artifact

- **some artifact is stimulated: a collection of systems, the whole system, or some piece or pieces of it**

Response

- **the activity undertaken as the result of the arrival of the stimulus**

Response measure

- **when the response occurs, it should be measurable in some fashion so that the requirement can be tested**

Quality Attribute
Availability

A property of software that it is there and ready to carry out its task when you need it to be.

Builds upon the concept of reliability by adding the notion of recovery

“Availability refers to the ability of a system to mask or repair faults such that the cumulative service outage period does not exceed a required value over a specified time interval.”

Availability is about minimizing service outage time by mitigating faults.

Dealing With Faults
(aka Defects, Errors, Bugs, …)

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.

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

Measuring 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:

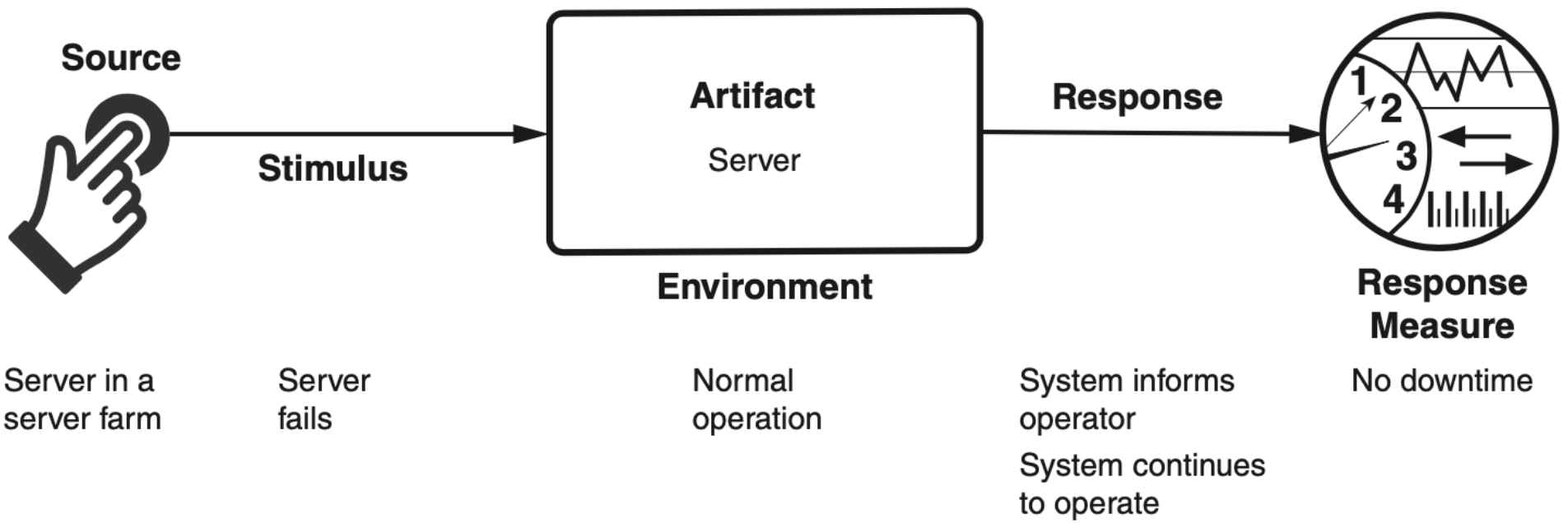
MTBF / (MTBF + MTTR)

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

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

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*.**



Goal of Availability Tactics

A failure occurs when the system no longer delivers a service consistent with its specification

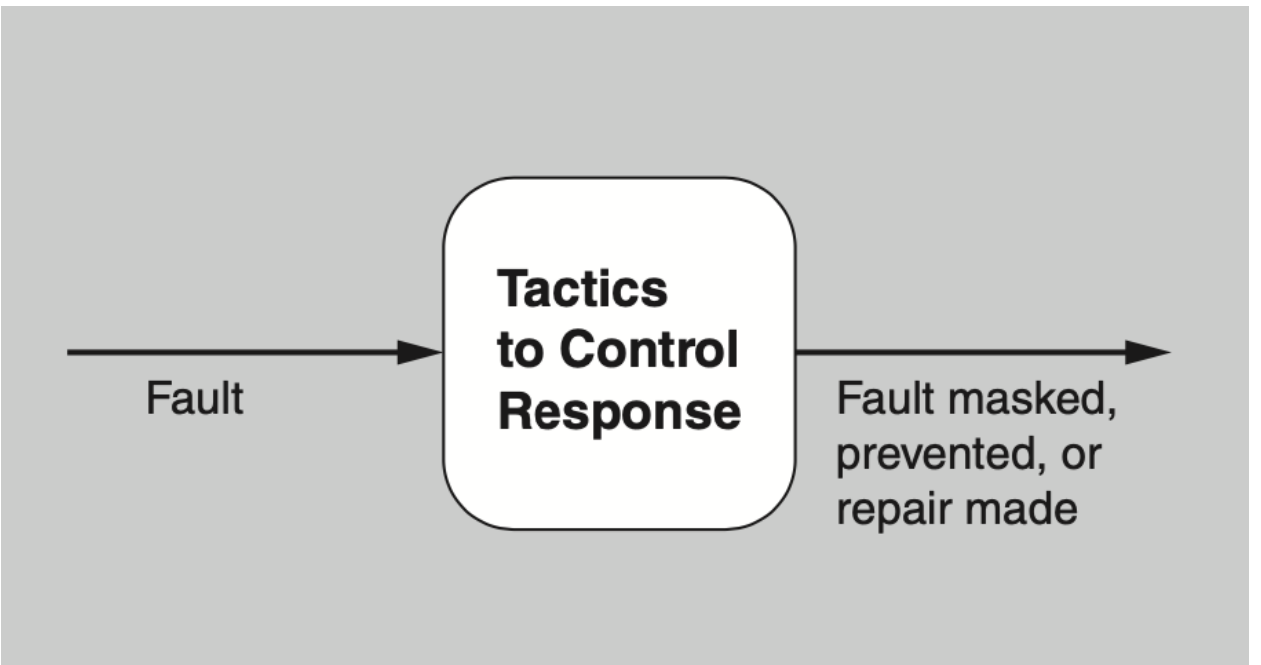
- **this failure is observable by the system's actors.**

A fault (or combination of faults) has the potential to cause a failure.

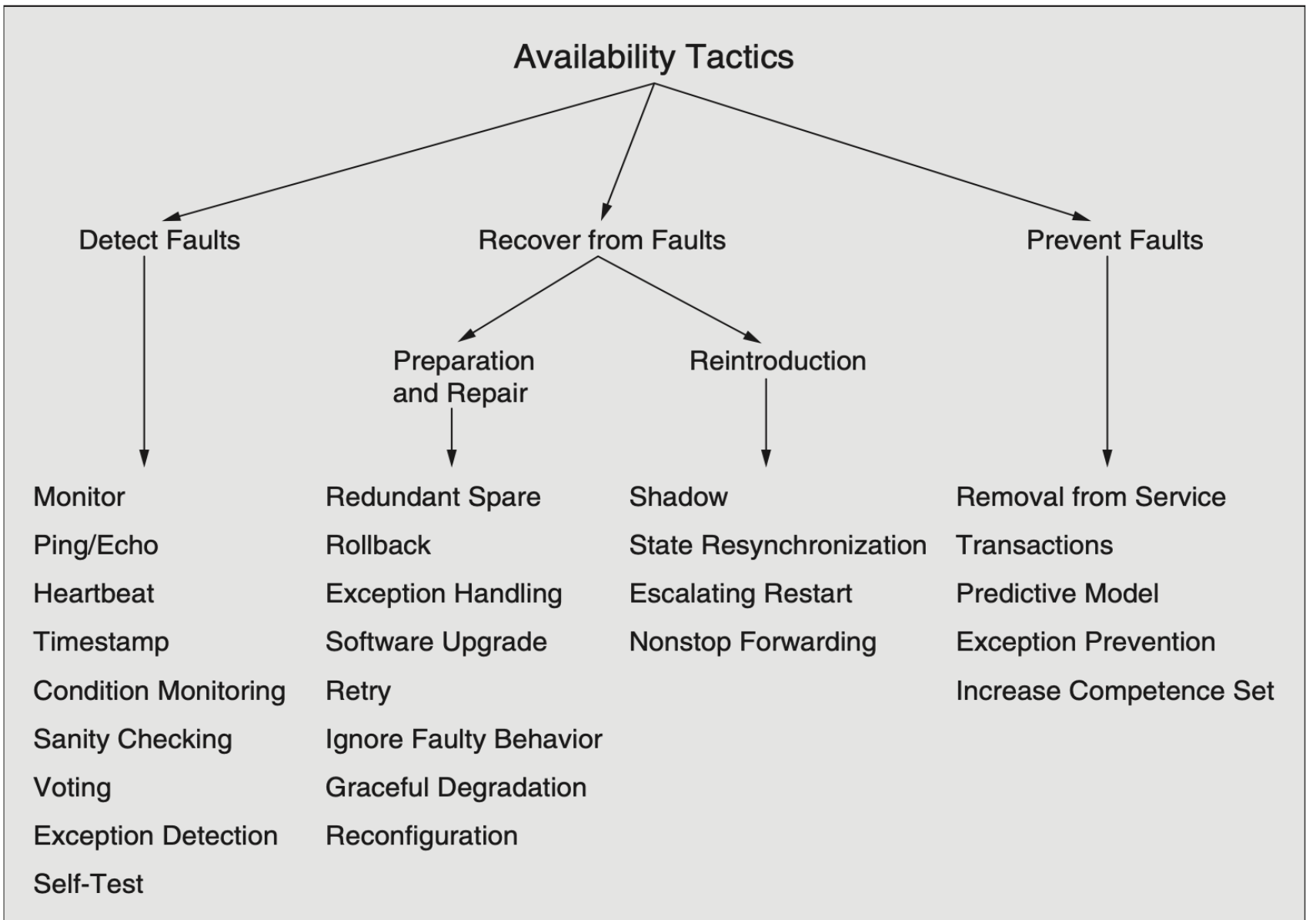
Availability tactics enable a system to endure faults so that services remain compliant with their specifications.

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



Quality Attribute

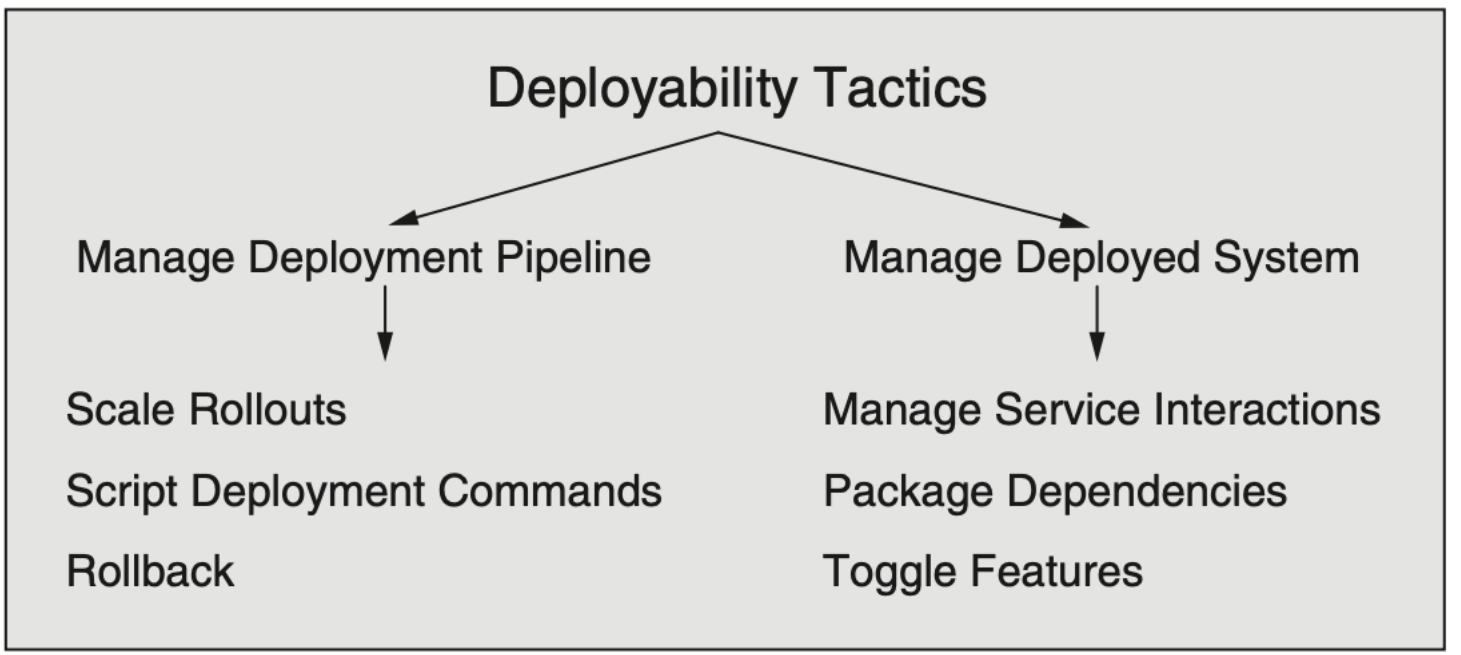
Deployability

Deployability **refers to a property of software indicating that it may be deployed—allocated to an environment for execution—within a predictable and acceptable amount of time and effort.**

Deployment is a process that starts with coding and ends with real users interacting with the system in a production environment.

If this process is fully automated—that is, if there is no human intervention—then it is called *continuous deployment*.

Deployability Tactics



Quality Attribute

Energy Efficiency

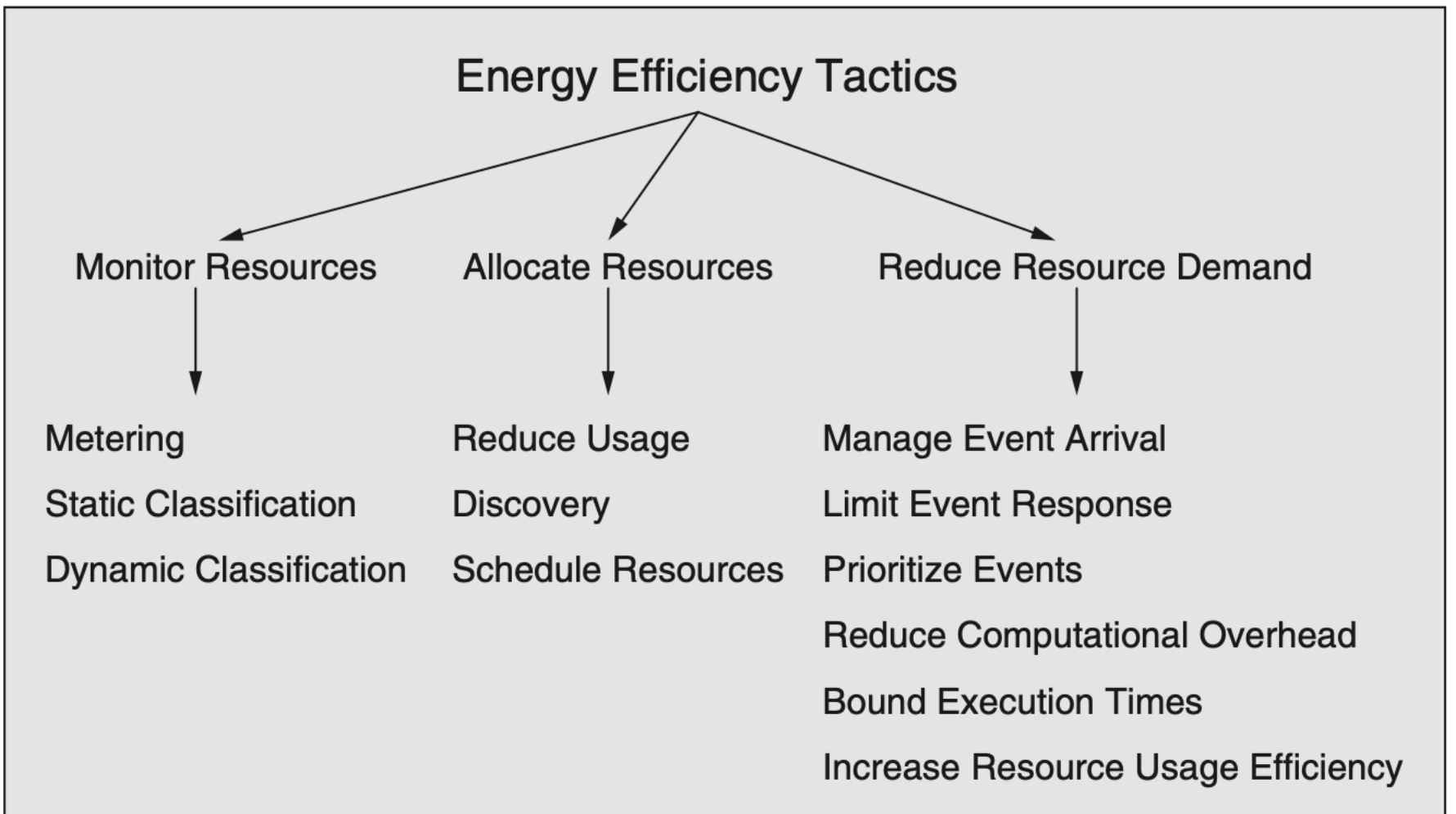
Energy used by computers used to be free and unlimited—or at least that's how we behaved.

Architects rarely worried about energy consumption of software in the past.

With the dominance of mobile devices, the increasing adoption of the IoT, and the ubiquity of cloud services, energy has become an issue that architects can no longer ignore.

The consumption of energy can be managed by the architect, as with any other quality attribute.

Energy Efficiency Tactics



Quality Attribute
Integrability

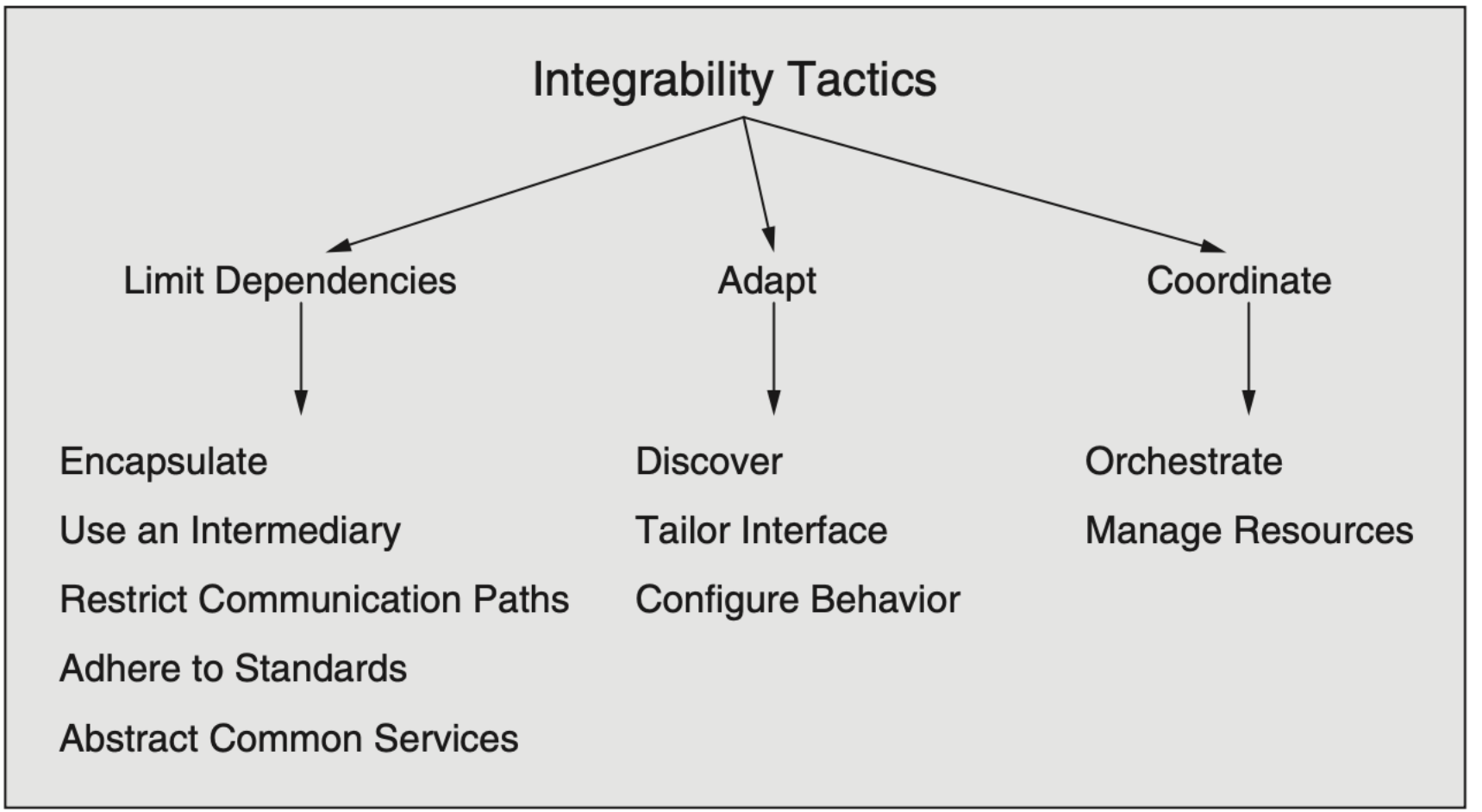
Software architects need to be concerned about more than just making separately developed components cooperate; they are also concerned with the *costs* and *technical risks* of integration tasks.

These risks may be related to schedule, performance, or technology.

Integration difficulty can be thought of as a function of

- **Size – the number of potential dependencies between the components and the system**
- **Distance – the difficulty of resolving differences at each of the dependencies**

Integrability Tactics



Quality Attribute
Modifiability

Modifiability is about change, and our interest in it centers on the cost and risk of making changes.

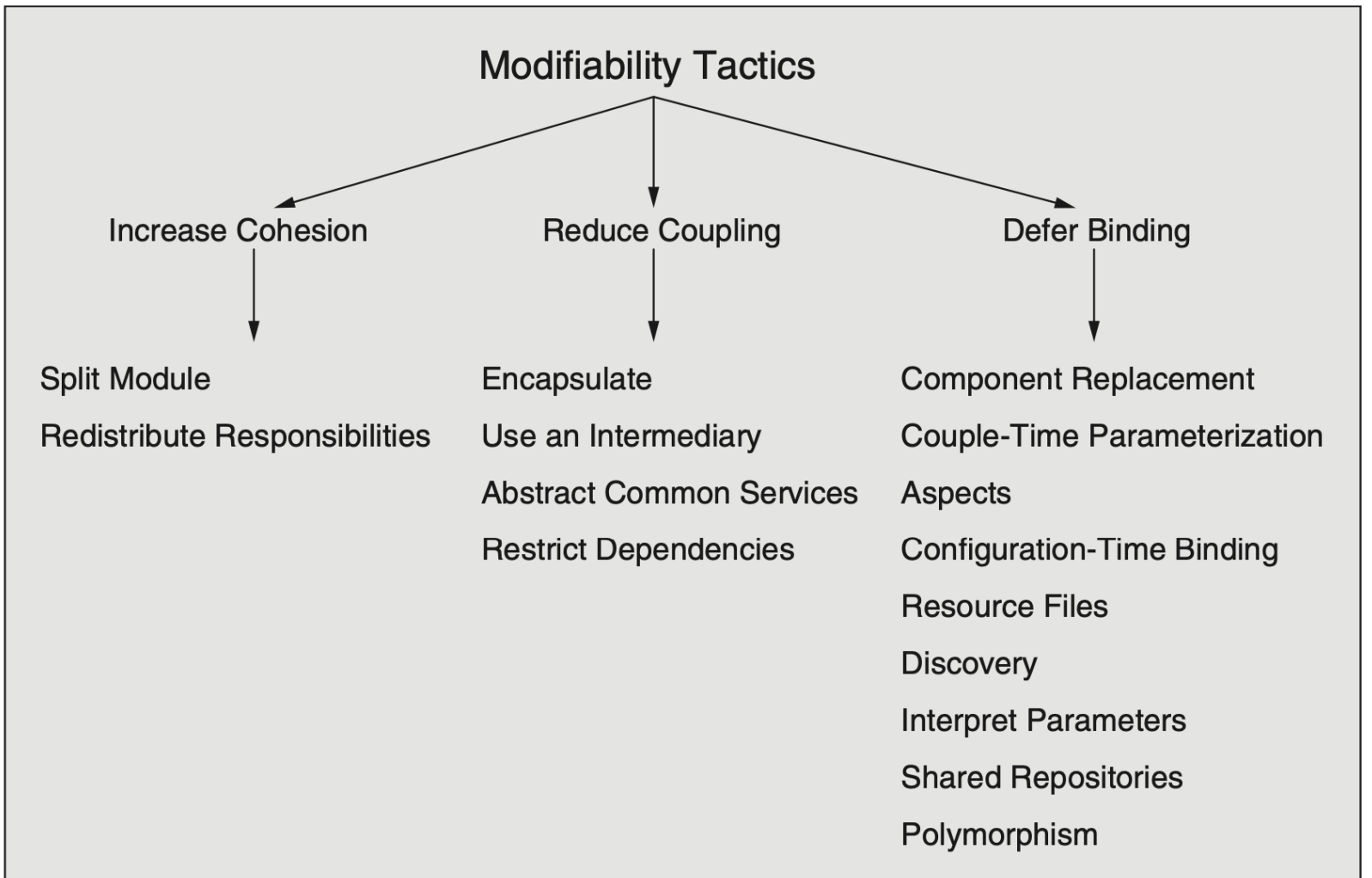
What can change?

What is the likelihood of the change?

When is the change made and who makes it?

What is the cost of the change?

Modifiability Tactics



Quality Attribute
Performance

It's about time and the software system's ability to meet timing requirements.

When events occur, the system, or some element of the system, must respond to them in time.

- **real-time**
- **hard real-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.

Measuring System Response

Latency

- **the time between the arrival of the stimulus and the system's response to it**

Deadlines

Throughput

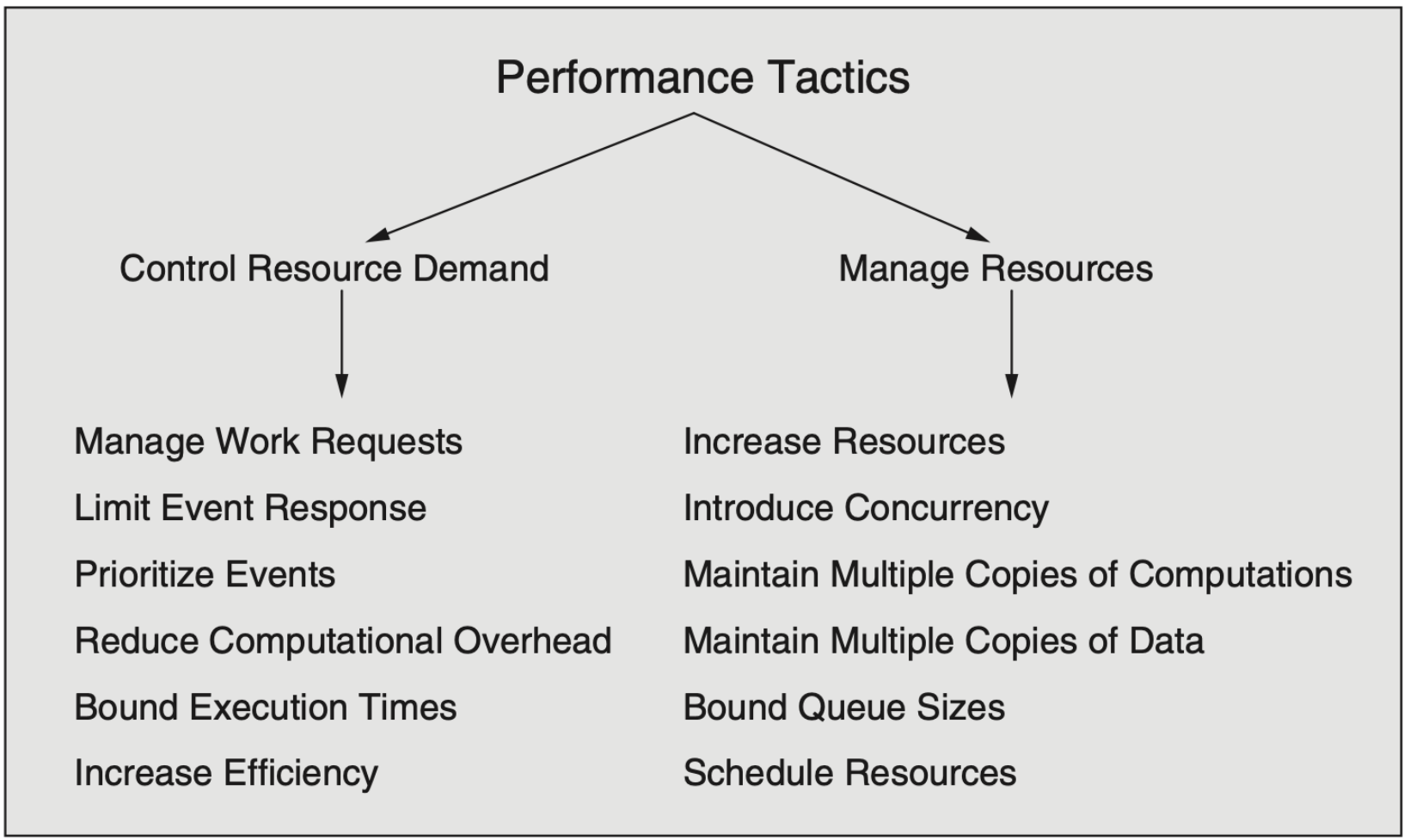
- **usually given as the number of transactions the system can process in a unit of time**

Jitter of the response

- **the allowable variation in latency**

The number of events not processed because the system was too busy to respond.

Performance Tactics



Quality Attribute
Safety

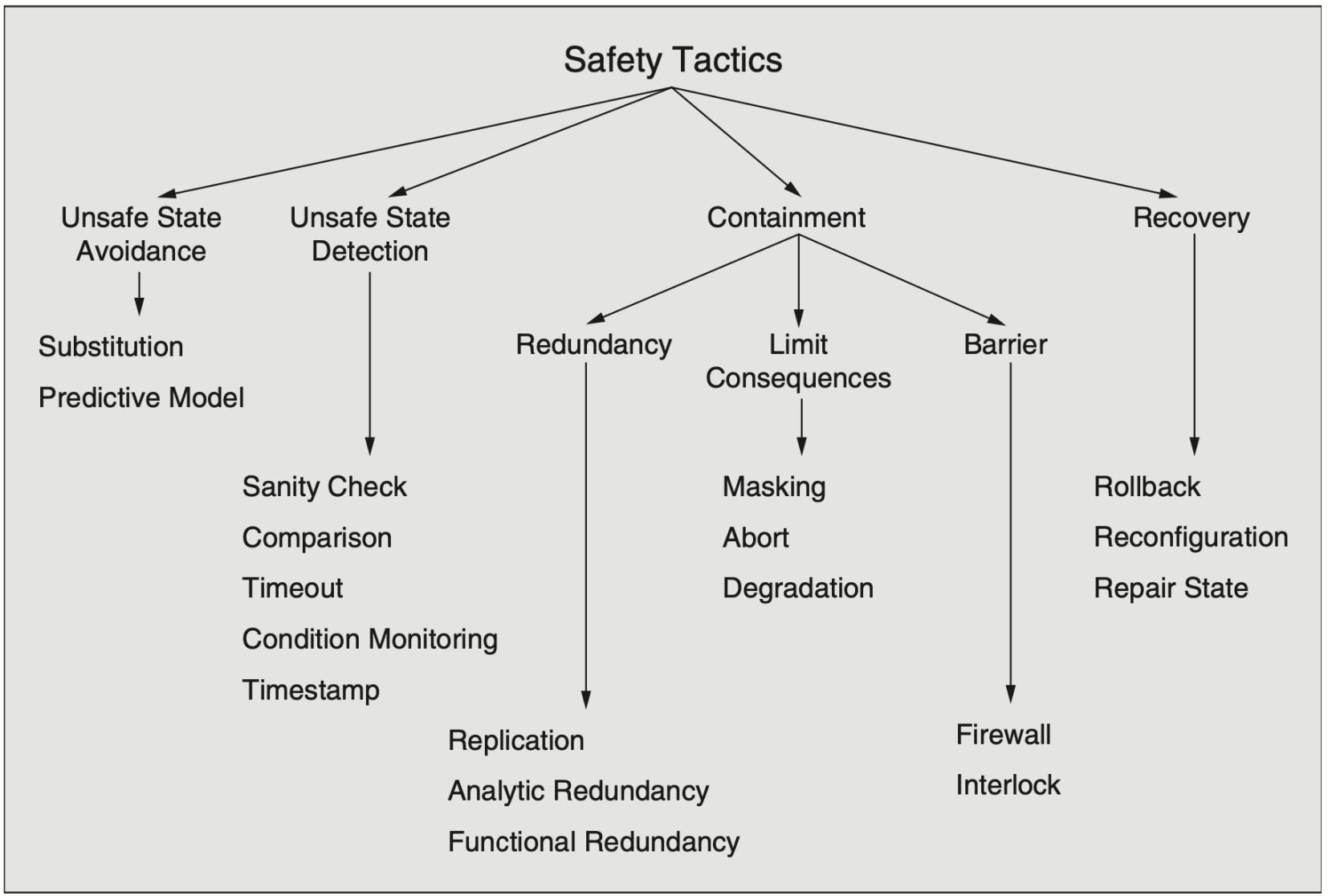
“Don’t kill anyone” should be a part of every software architect’s mission statement.

Safety is concerned with a system’s ability to avoid straying into states that lead to damage, injury, or loss of life, which can be caused by a variety of factors

- **Omissions**
- **Commission**
- **Timing**
- **Problems with system values**
- **Sequence omission and commission**
- **Out of sequence**

Safety is concerned with detecting and recovering from these unsafe states to prevent or minimize harm.

Safety Tactics



Quality Attribute

Security

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.

- **an unauthorized attempt to access data or services**
- **an unauthorized attempt to modify data**
- **intended to deny services to legitimate users**

A Simple Approach to Security

Confidentiality

- **data or services are protected from unauthorized access**
  - **a hacker cannot access your income tax returns on a government computer**

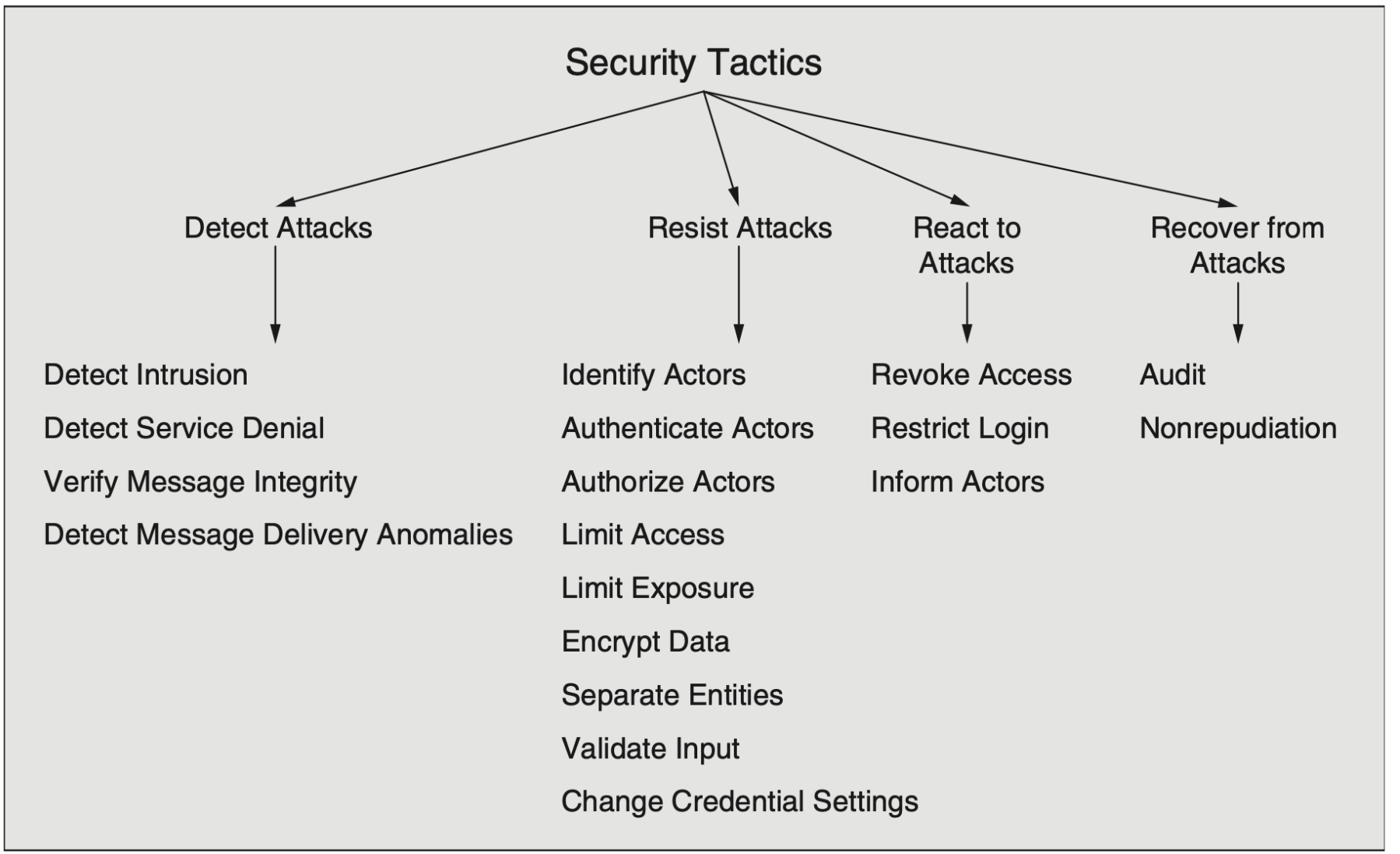
Integrity

- **data or services are not subject to unauthorized manipulation**
  - **your grade has not been changed since your instructor assigned it**

Availability

- **the system will be available for legitimate use**
  - **a denial-of-service attack won't prevent you from ordering book from an online bookstore**

Security Tactics



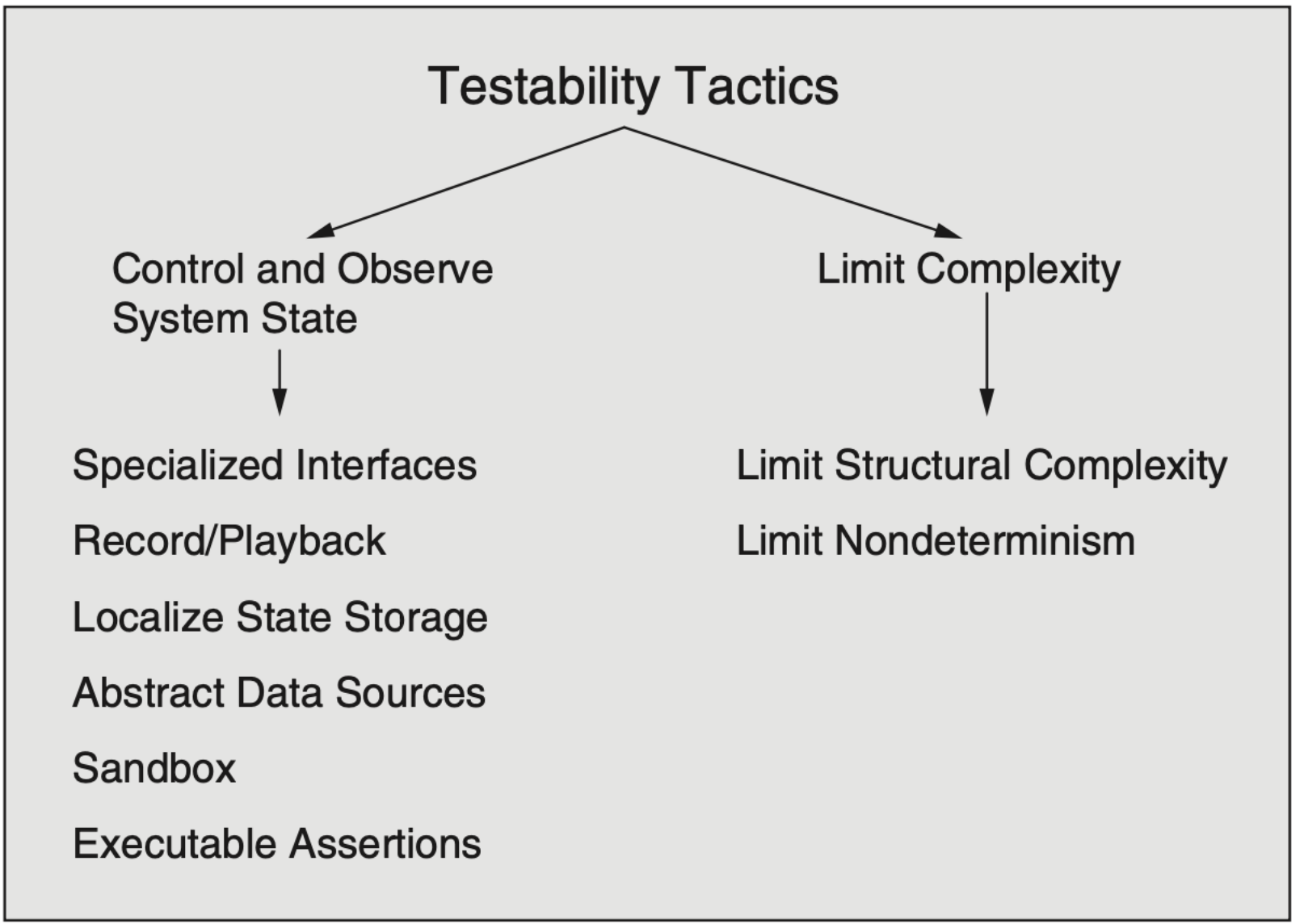
Quality Attribute
Testability

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.

Intuitively, a system is testable if it "reveals" its faults easily.

Testability Tactics



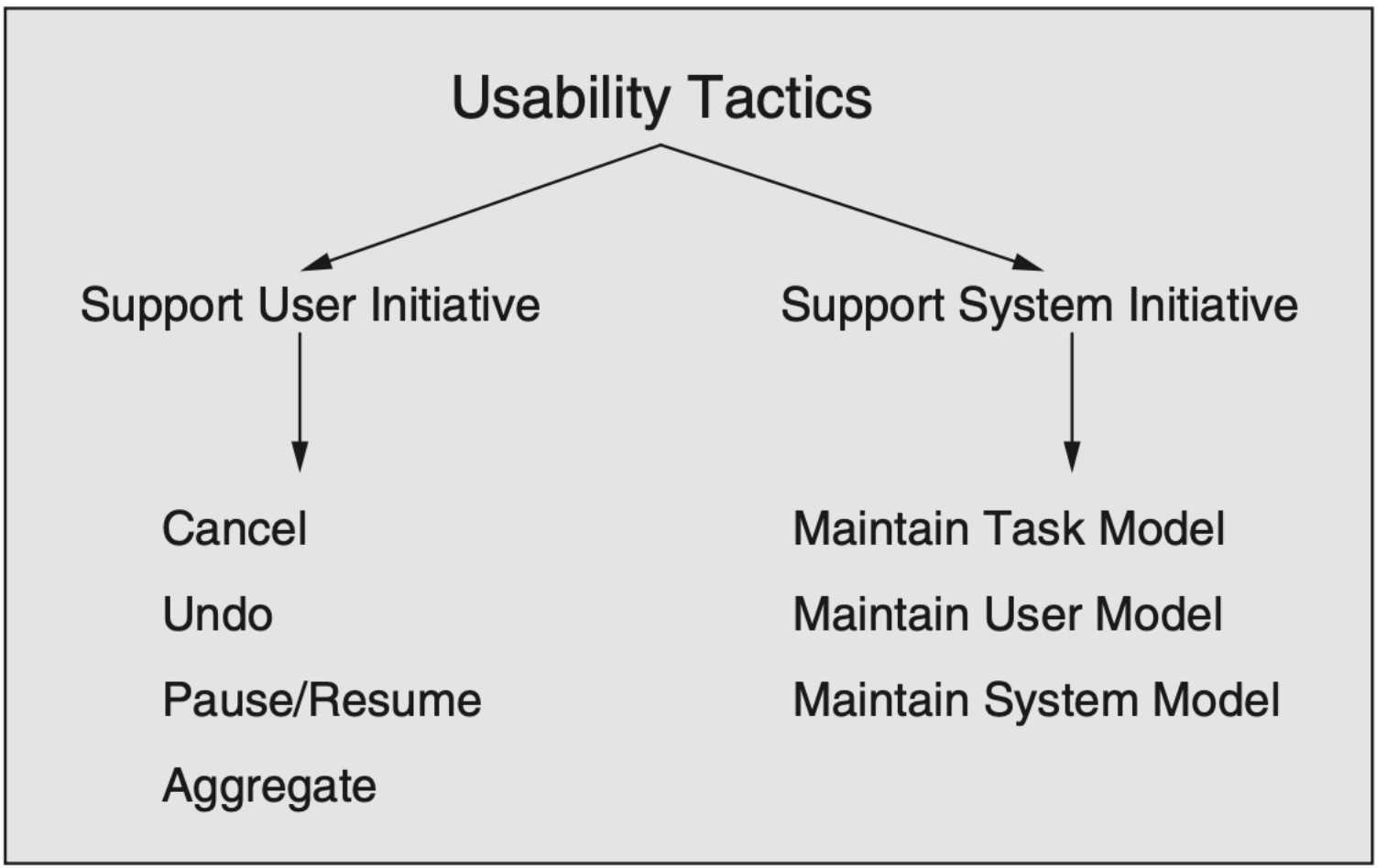
Quality Attribute
Usability

Concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides.

Usability comprises

- **Learning system features.**
- **Using a system efficiently.**
- **Minimizing the impact of errors.**
- **Adapting the system to user needs.**
- **Increasing confidence and satisfaction.**

Usability Tactics



ISO/IEC 25010 Systems and Software Quality Requirements and Evaluation (SQuaRE)

Functional suitability

- functional completeness
- functional correctness
- functional appropriateness

Performance efficiency

- time behavior
- resource utilization
- capacity

Compatibility

- coexistence
- interoperability

Usability

- appropriateness recognizability
- learnability
- operability
- user error prediction
- user interface aesthetics
- accessibility

Reliability

- maturity
- availability
- fault tolerance
- recoverability

Security

- confidentiality
- integrity
- nonrepudiation
- accountability
- authenticity

Maintainability

- modularity
- reusability
- analyzability
- modifiability
- testability

Portability

- adaptability
- installability
- replaceability

Dealing with a New Quality Attribute

Capture scenarios

Assemble design approaches

Model the new quality attribute

Assemble a set of tactics

Construct design checklists

Architectural Pattern

Is a package of design decisions that is found repeatedly in practice

Has known properties that permit reuse

Describes a class of architectures

Inspired by Christopher Alexander's A Pattern Language: Towns, Buildings, Construction (1977).

The Gang of Four (GoF): E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1994.

Discovering Patterns

Patterns are by definition found in practice

- **one does not invent them**
- **one discovers them**

Patterns spontaneously emerge in reaction to environmental conditions

- **as long as conditions change, new patterns will emerge**

Relationships Between Tactics and Patterns

Tactics are the "building blocks" of design from which architectural patterns are created.

Most patterns are constructed from several different tactics.

- **these tactics might all serve a common purpose**
- **they are often chosen to promote different quality attributes**

A pattern is a general solution.

A documented pattern is underspecified with respect to applying it in a specific situation.

Patterns Establish A Relationship

A context

- **A recurring, common situation in the world that gives rise to a problem.**

A problem

- **outlines the problem and its variants**
- **describes any complementary or opposing forces**
- **includes quality attributes that must be met**

A solution

- **appropriately abstracted**
- **describes the architectural structures that solve the problem**
- **how to balance the forces at work**

The Solution for a Pattern

A set of element types

- **data repositories, processes, objects, …**

A set of interaction mechanisms or connectors

- **method calls, events, message bus, …**

A topological layout of the components

A set of semantic constraints covering topology, element behavior, and interaction mechanisms

Complex systems exhibit multiple patterns at once.

Architectural Patterns Listing
(SAiP 3rd edition)

Module patterns
- **layered**

Component-and-connector patterns
- **broker**
- **model-view-controller**
- **pipe-and-filter**
- **client-server**
- **peer-to-peer**
- **service-oriented architecture**
- **publish-subscribe**
- **shared-data**

Allocation patterns
- **map-reduce**
- **multi-tier**

Architectural Patterns Overview

Layered pattern

- **defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers.**
  - **usually shown graphically by stacking boxes representing layers on top of each other**

Broker pattern

- **defines a runtime component, called a broker, that mediates the communication between a number of clients and servers**

Model-view-controller pattern

- breaks system functionality into three components: a model, a view, and a controller that mediates between the model and the view
  - model is a representation of the application data or state, and it contains (or provides an interface to) application logic
  - view is a user interface component that either produces a representation of the model for the user or allows for some form of user input or both
  - controller manages the interaction between the model and the view, translating user actions into changes to the model or changes to the view

Pipe-and-filter pattern

- transforms data from a system's external inputs to its external outputs through a series of transformations performed by its filters, which are connected by pipes

Client-server pattern

- **clients initiate interactions with servers, invoking services as needed from those servers and waiting for the results of those requests**

Peer-to-peer pattern

- **computation is achieved by cooperating peers that request service from, and provide services to, one another across a network**

Service-oriented-architecture (SOA) pattern

- **computation is achieved by a set of cooperating components that provide and/or consume services over a network**
  - **the computation is often described using a workflow language**

Publish-subscribe pattern

- components publish and subscribe to events
  - when an event is announced by a component, the connector infrastructure dispatches the event to all registered subscribers

Shared-data pattern

- communication between data accessors is mediated by a shared data store
  - control may be initiated by the data accessors or the data store
  - data is made persistent by the data store

Map-reduce pattern

- provides a framework for analyzing a large distributed set of data that will execute in parallel on a set of processors
  - parallelization allows for low latency and high availability
  - the map performs the *extract* and *transform* portions of the analysis
  - the reduce performs the *loading* of the results
  - *extract-transform-load* is sometimes used to describe the functions of the map and reduce

Multi-tier pattern

- the execution structures of many systems are organized as a set of logical groupings of components termed *tiers*
  - tiers may be based on a variety of criteria, such as the type of component, sharing the same execution environment, or having the same runtime purpose

Architectural Pattern Example

Layered Pattern

Context

- **all complex systems experience the need to develop and evolve portions of the system independently**
- **need a clear and well-documented separation of concerns**

Problem

- **The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse.**

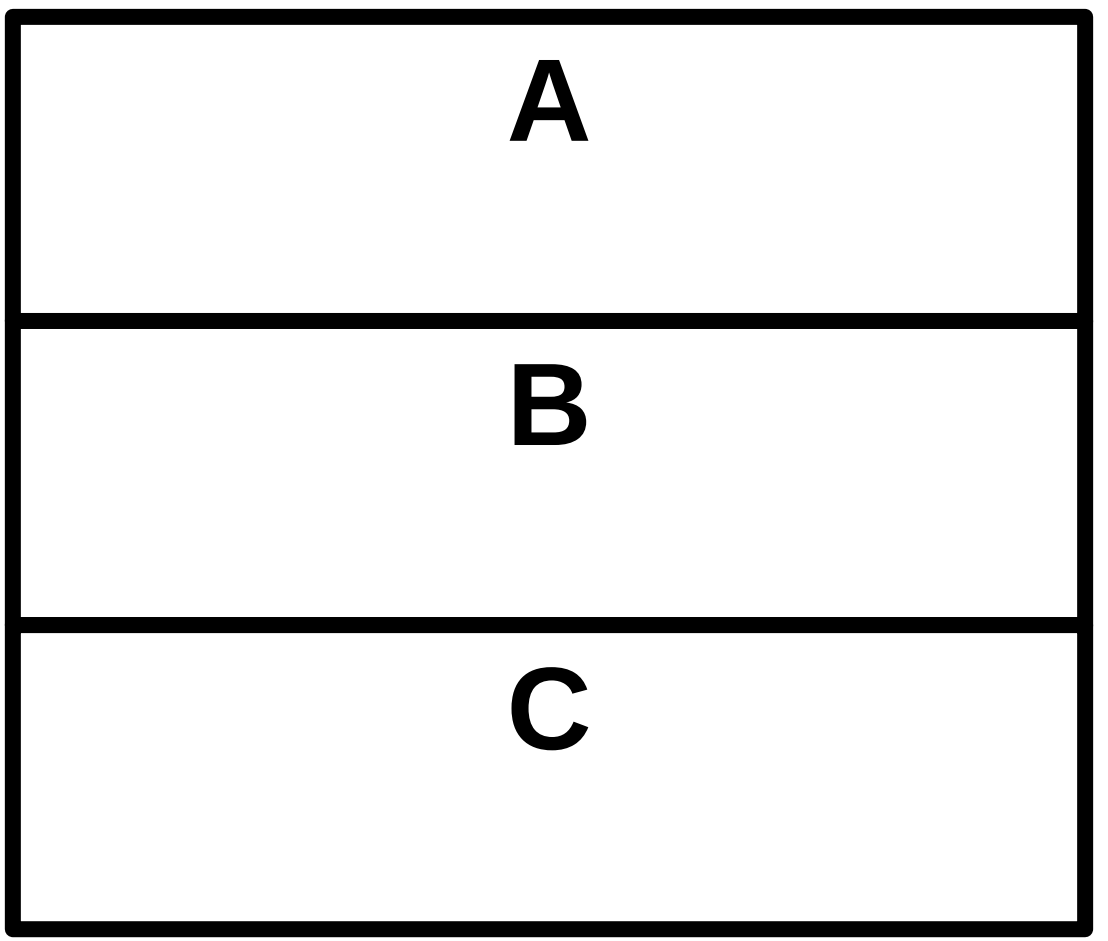
Solution

- **divide the software into units called layers**
- **each layer is a grouping of modules that offers a cohesive set of services**
- **completely partition a set of software**
- **a public interface**
- **relations between layers must be unidirectional**
  - **if (A,B) is in this relation, we say that the implementation of layer A is allowed to use any of the public facilities provided by layer B**

Pattern Almanac 2000 by L. Rising lists over 100 patterns that are variants of, or related to, Layers.

Stack of Boxes Notation

Allowed-to-use relation reads from top down



Finer Points of Layers
No Using Above

The most important point about layering is that a layer isn't allowed to *use* any layer above it.

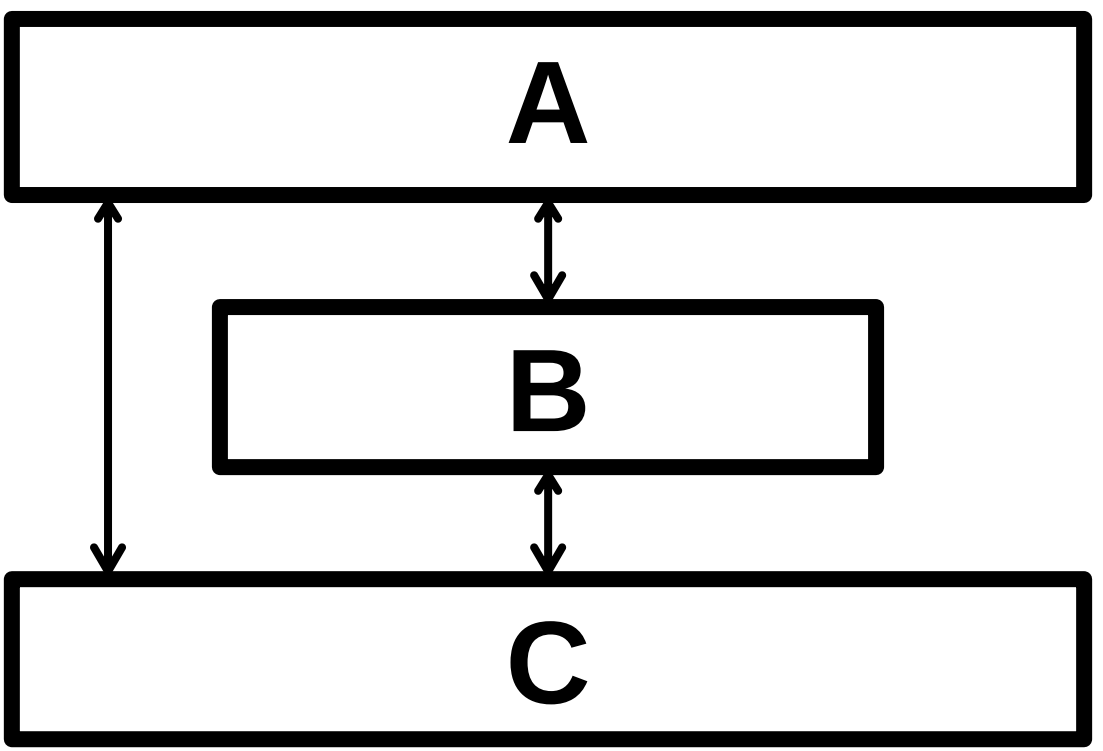
A module "uses" another module when it depends on the answer it gets back.

- **a layer is allowed to make upward calls, as long as it isn't expecting an answer**
- **this is how the common error-handling scheme of callbacks works**

Finer Points of Layers
Arbitrary Allowed-to-Use

Any old set of boxes stacked on top of each other does not constitute a layered architecture.

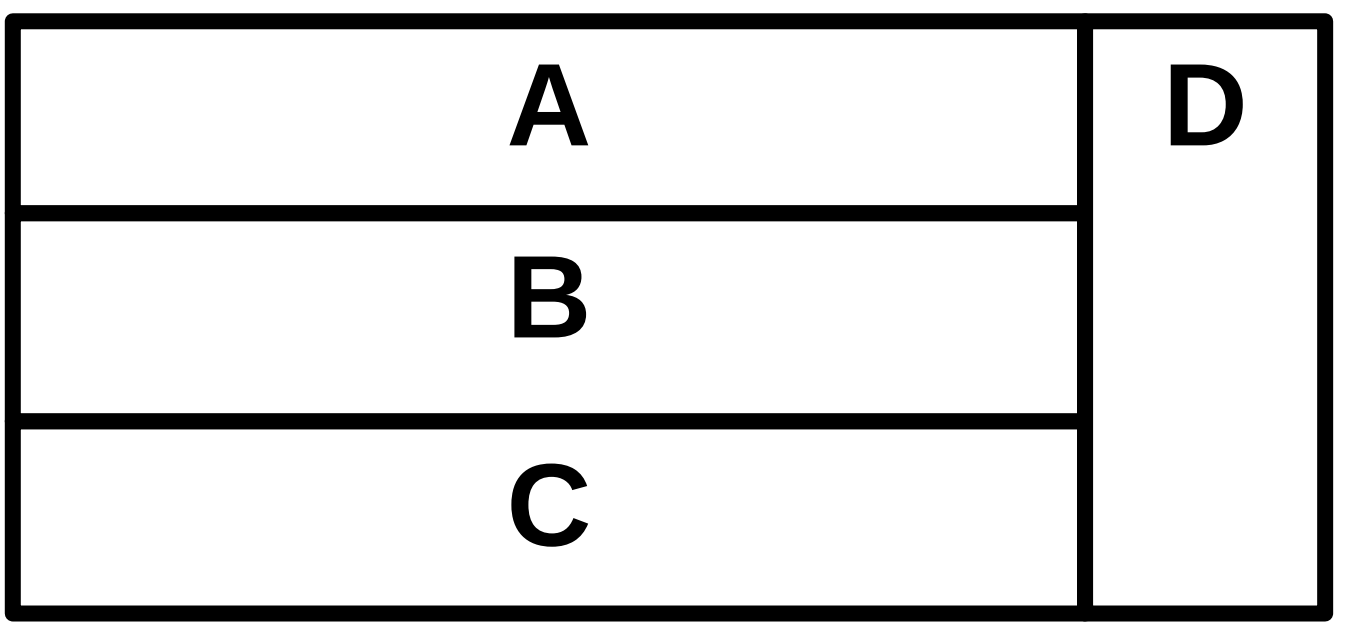
- **avoid arrows (allowed to use)**



Finer Points of Layers
Sidecars

“Sidecars” may contain common utilities (sometimes imported).

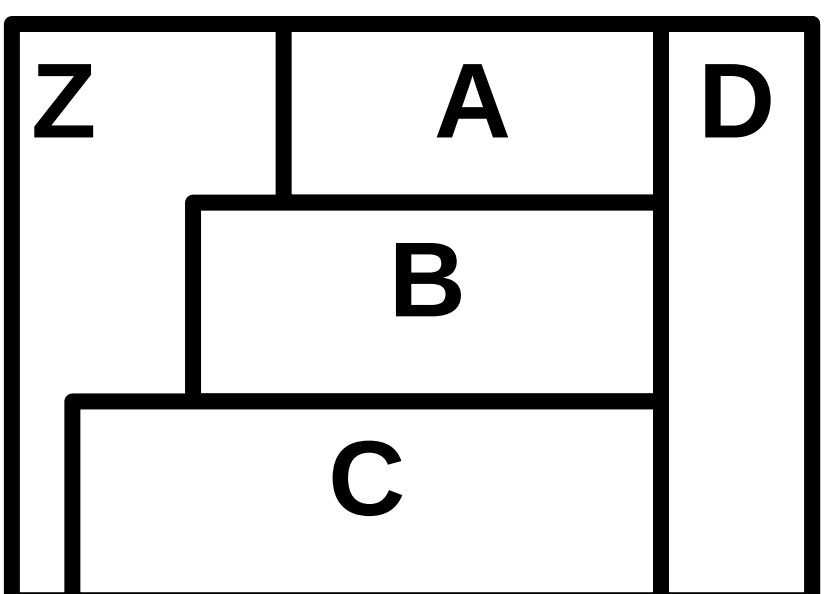
- **without a key, are you sure?**



Finer Points of Layers
Bridging

It is impossible to look at a stack of boxes and tell whether layer bridging is allowed or not.

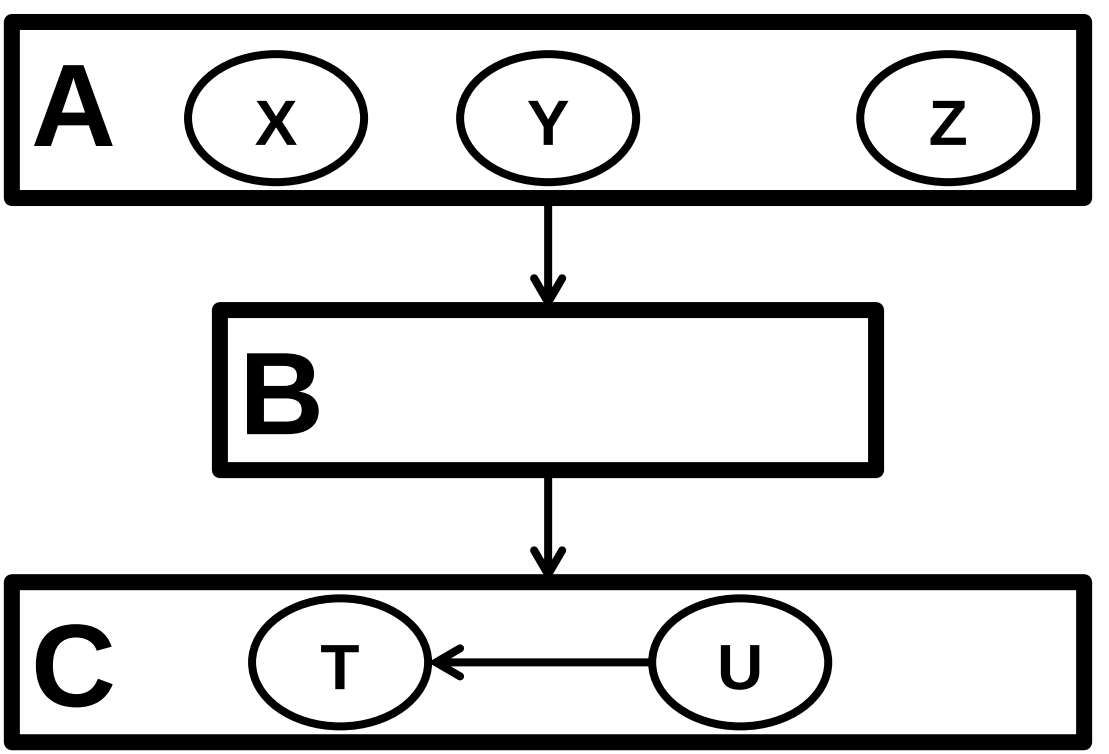
- **add stairsteps or vertical layers to your notation**
- **adding a key is essential!**



Finer Points of Layers
Segments

Segments may denote a finer-grained decomposition of the modules.

- **specify what usage rules are in effect among the segments**



Advantages of the Layered Pattern

Support design based on increasing levels of abstraction

Support enhancement

- **affect at most two other layers**

Support reuse

- **standard interfaces with multiple implementations**

Weaknesses of the Layered Pattern

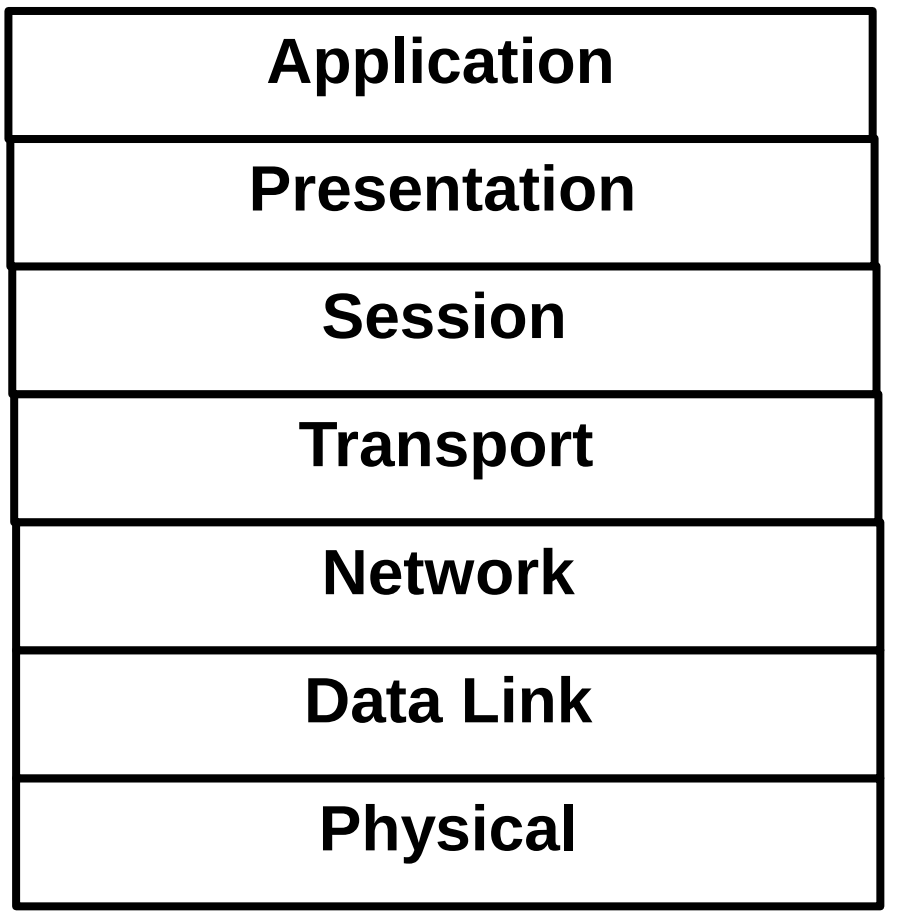
Adds up-front cost and complexity

Contribute a performance penalty

Bridging may prevent meeting portability and modifiability goals

Example of the Layered Pattern

ISO Open System Interconnection (OSI)



Architectural Pattern Example

Model-View-Controller (MVC) Pattern

Context

- **user interface (UI) most frequently accessed and changed component of an interactive application**
- **users frequently like to have multiple perspectives (views) into the system**

Problem

- **in a large complex system, separate the UI design from the rest of the system**
- **allow changes to the interface with minimal impact on the rest of the system**

Solution

- **MVC pattern separates application functionality into three kinds of components**
  - **Model – internal state of the application**
  - **View – external representation of the model**
  - **Controller – coordinates updates of the view in response to user input or model changes**

Originally formulated in the late 1970s at Xerox PARC

Central ideas

- ***code reusability***
- ***separation of concerns***

MVC Elements

Model
- **interacts with the data model of the application**
- **notifies the view when the state is updated allowing the view to change**

View
- **presentation layer for the application**
- **gets information from the model to update the presentation**

Controller
- **defines the way UI reacts to user input**
- **sends commands to the view to change the view's presentation**
- **sends commands to the model to update the state of the model**

Model-View-Controller Overview

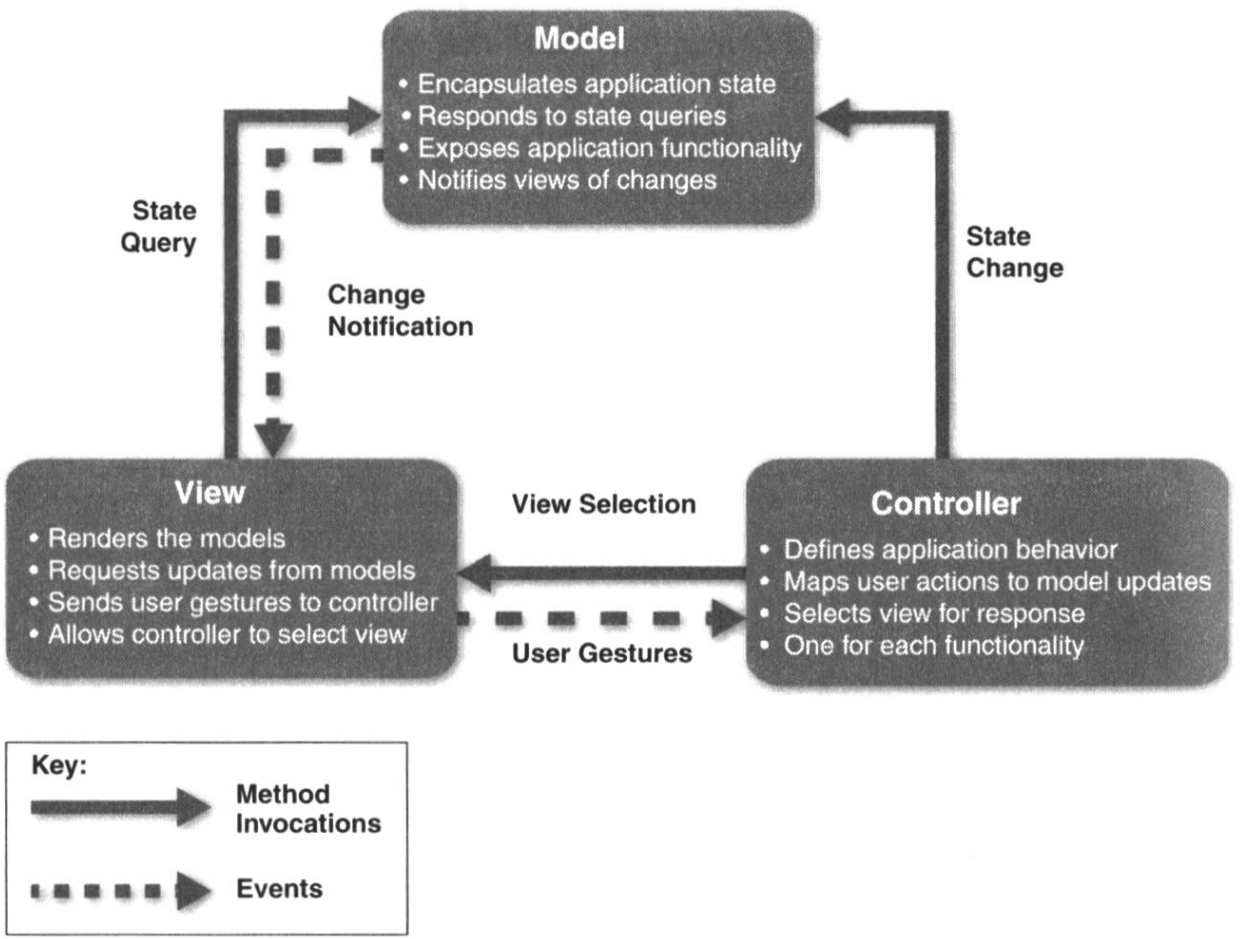


FIGURE 13.7 The model-view-controller pattern

Examples of the MVC Pattern

Video games

- **MVC helps separate game, view and input logic**
- **allows change to code and ideas with ease**
- **each component can be easily added or removed without affecting other components**
- **allows reuse of code simply**

Web applications

- **PHP-based MVC web frameworks**
  - **CakePHP**
  - **CodeIgniter**
- **Java-based MVC web frameworks**
  - **Apache Struts**
  - **Wicket**

Advantages of the MVC Pattern

Increase flexibility and reuse

- **separation of concerns**
- **reduced coupling**

Easily incorporate multiple views

Promotes testability through defined interfaces

Weaknesses of the MVC Pattern

Some fundamental complexity

- **perhaps too complex for simple applications**

Variance of the pattern among tools can be substantial

Two Perspectives

To make a pattern work…

Inherent quality attribute tradeoffs that the pattern makes

- **Patterns exist to achieve certain quality attributes, and we need to compare the ones they promote (and the ones they diminish) with our needs.**

Other quality attributes that the pattern isn't directly concerned with

- **but it affects…**
- **which are important in our application**

Pattern	Modifiability									
	Increase Cohesion		Reduce Coupling					Defer Binding Time		
	Increase Semantic Coherence	Abstract Common Services	Encapsulate	Use a Wrapper	Restrict Comm. Paths	Use an Intermediary	Raise the Abstraction Level	Use Runtime Registration	Use Startup-Time Binding	Use Runtime Binding
Layered	X	X	X		X	X	X			
Pipes and Filters	X		X		X	X			X	
Blackboard	X	X			X	X	X	X		X
Broker	X	X	X		X	X	X	X		
Model View Controller	X		X			X				X
Presentation Abstraction Control	X		X			X	X			
Microkernel	X	X	X		X	X				
Reflection	X		X							

Using Tactics Together

Decided to employ ping/echo to detect failed components →

Security

- **How to prevent a ping flood attack?**

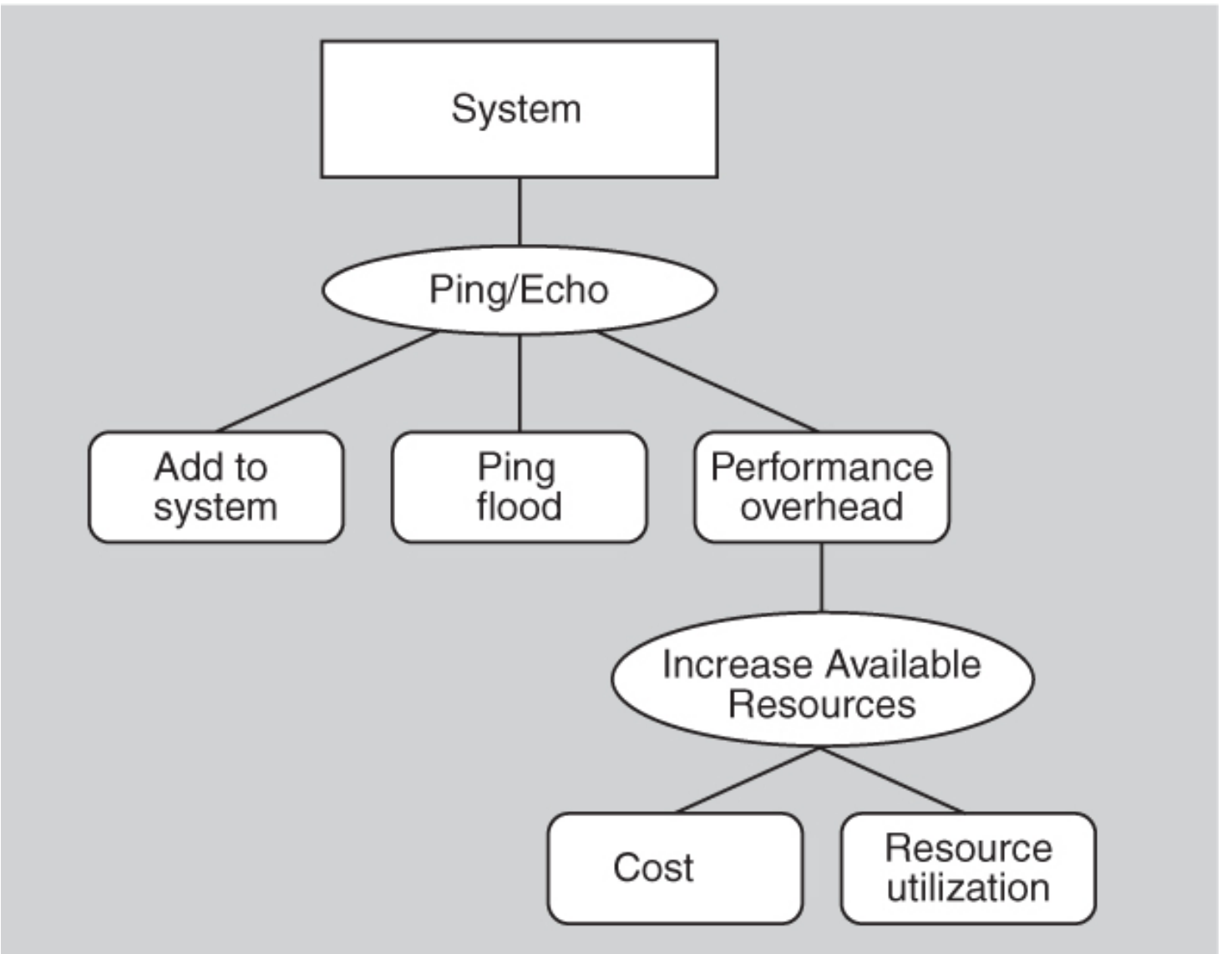
Performance

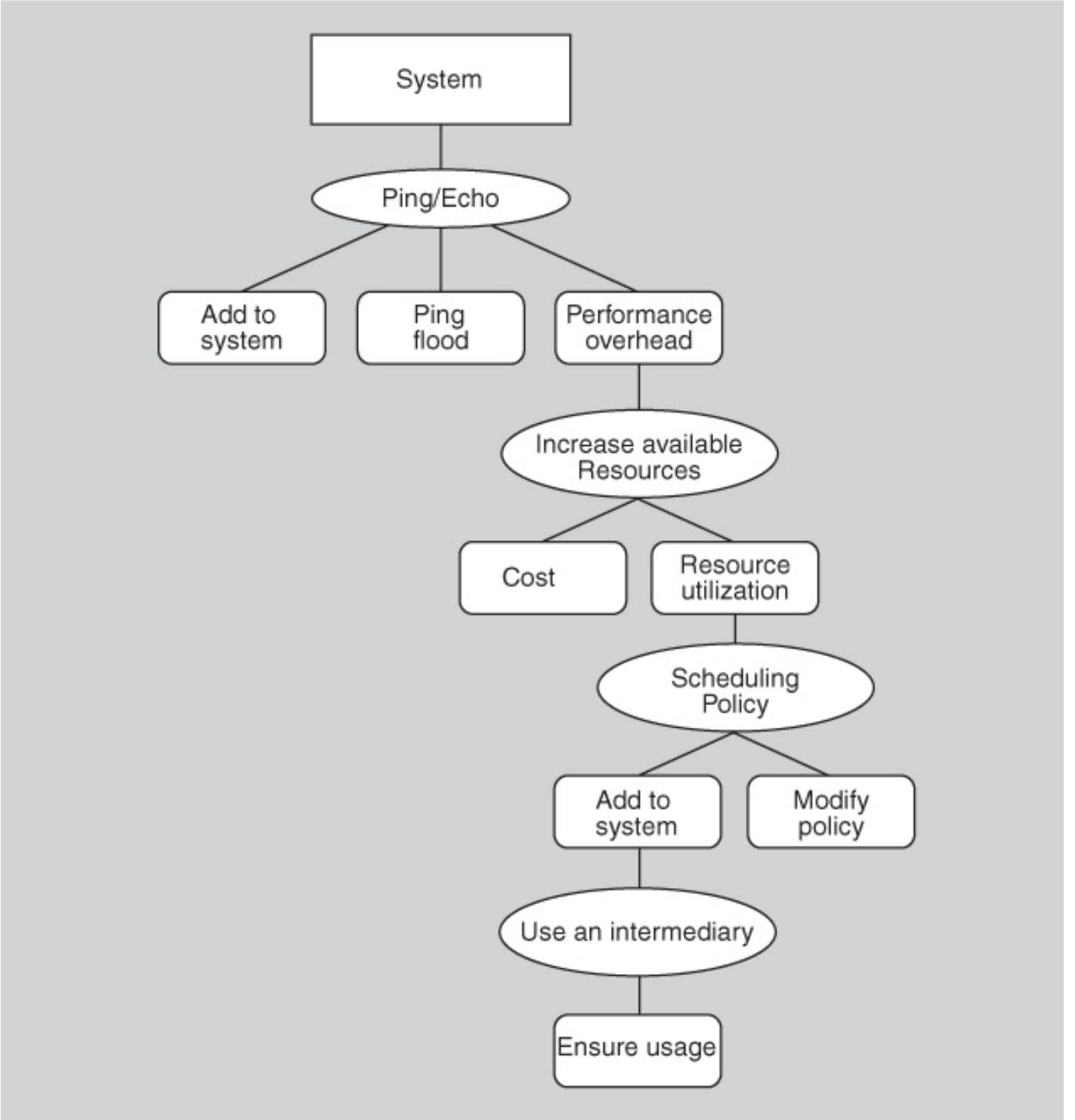
- **How to ensure that the performance overhead of ping/echo is small?**

Modifiability

- **How to add ping/echo to the existing architecture?**

Focus on Performance Tradeoff




System
Ping/Echo
Add to system
Ping flood
Performance overhead
Increase available Resources
Cost
Resource utilization
Scheduling Policy
Add to system
Modify policy
Use an intermediary
Ensure usage

Recognizing a Good Architecture

There is no such thing as an inherently good (or bad) architecture – architecture should fit some purpose.

- ***Arguable point … but note the hard tradeoff sometimes made between performance and modifiability***

Architectures can only be evaluated in light of specific stated goals.

Bass, Clements, and Kazman identify rules of thumb for designing architectures.

- **process recommendations**
- **structural recommendations**

Architectural Process Recommendations

The architecture should be the product of a single architect or a small group of architects with an identified technical leader.

- **There should be a strong connection between the architect(s) and the development team.**

The architect (or architecture team) should, on an ongoing basis, base the architecture on a prioritized list of well-specified quality attribute requirements.

- **Tradeoffs will always occur.**

The architecture should be documented using views.

- **The views should address the concerns of the most important stakeholders in support of the project timeline.**

The architecture should be evaluated for its ability to deliver the system's important quality attributes.

The architecture should lend itself to incremental implementation.

- **Cockburn's walking skeleton and incremental rearchitecture approach**

Structural Rules of Thumb

The architecture should feature well-defined modules whose functional responsibilities are assigned on the principles of information hiding and separation of concerns.

- **encapsulation**
- **well-defined interfaces**
- **information hiding**

Quality attributes should be achieved using well-known architectural patterns and tactics specific to each attribute.

The architecture should never depend on a particular version of a commercial product or tool.

Modules that produce data should be separate from modules that consume data.

Don’t expect a one-to-one correspondence between modules and components.

- **concurrency**
- **multi-threaded systems**

Every process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.

The architecture should feature a small number of ways for components to interact.

- **do things in the same way throughout (consistency)**
- **low coupling**

The architecture should contain a specific (and small) set of resource contention areas, the resolution of which is clearly specified and maintained.

- **high cohesion**
- **separation of concerns**

Questions?

Dr. Mark C. Paulk
University of Texas at Dallas
ECSS 3.610, EC31
800 W. Campbell Road
Richardson, TX 75080-3021

Mark.Paulk@utdallas.edu
Mark.Paulk@ieee.org

https://personal.utdallas.edu/~mcp130030/