AARHUS UNIVERSITET

Software Architecture in Practice

Architectural Design

The bottom line

Bertrand Meyer:

... once everything has been said,

software is defined by code.

Or – in other words:

- Architectural views, UML, quality attribute scenarios don’t pay the bills...

What do we do?

We have identified that our architecture should strike a balance between qualities A, B, and C – in various scenarios.

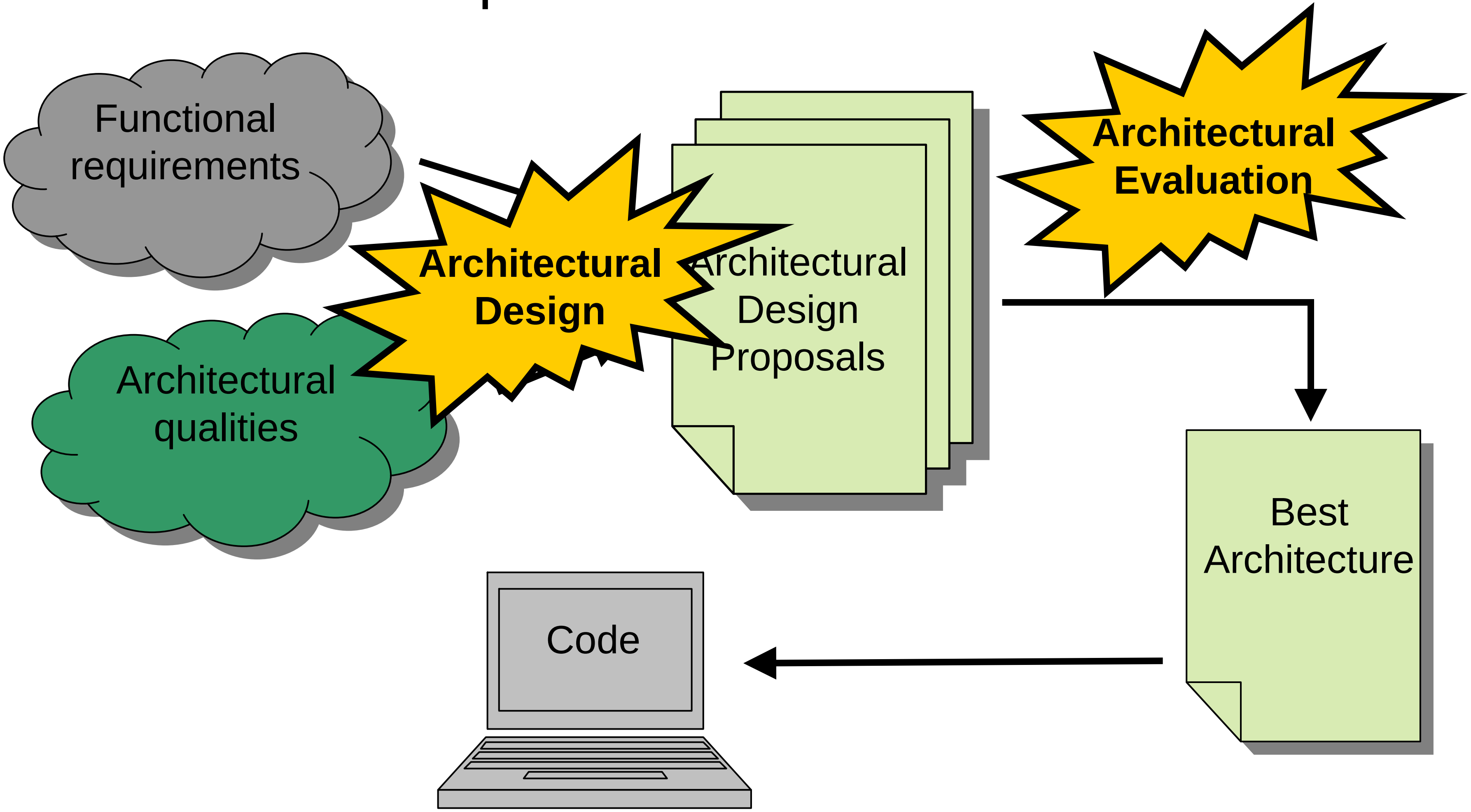
- and - Conceptual integrity, Correctness and completeness, and Buildability

Question: How do we then use this information to guide the design ???

Architectural Design ?

Architectural design

The idealized process



Architectural Design

- The process of designing a software architecture that meets quality attribute requirements
  - And enables implementation of functional requirements
- Characteristics of the process
  - Creative
  - Iterative (and incremental)
    - Functional decomposition
    - Quality decomposition
  - Experimental
    - Architectural prototyping
  - Based on experience
    - This lesson

Overview

Architectural Styles

- A vocabulary of large-scale structure

Architectural Patterns

- Name, Problem, Solution, Consequences
- Patterns = Styles?

Tactics

- Surgical bits and pieces

Architectural Style

Architectural Style

An architectural style is a description of component types and a pattern of their runtime control and/or data transfer.

– defines constraints on component types and interaction patterns

– thereby delimits/spans a set of architectures

– (also called architectural pattern ☺)

Ex.: Client-Server

The parts of a ’style’

Parts of a style

- A set of *component* types with a given role/functionality
- A *topology* of relations (usually runtime relations)
- A set of *connectors* (RMI, socket, memory, etc.) that handle communication, coordination or collaboration.
- A set of *semantic constraints*
  - i.e. what can components/connectors do or not do?

Exercise: Client-server

Component types?

- categories of components

Topology?

- (“the landscape” / set of relations)

Connectors?

- what are the carriers of data- and control flow?

Semantic constraints?

- what rule must the components/connectors obey?

Why is it interesting?

Why are architectural styles / patterns interesting and important?

Why is it interesting?

Because they

describe architectures with specific qualities

…and

- document it
- provides a vocabulary

Classification in Bass

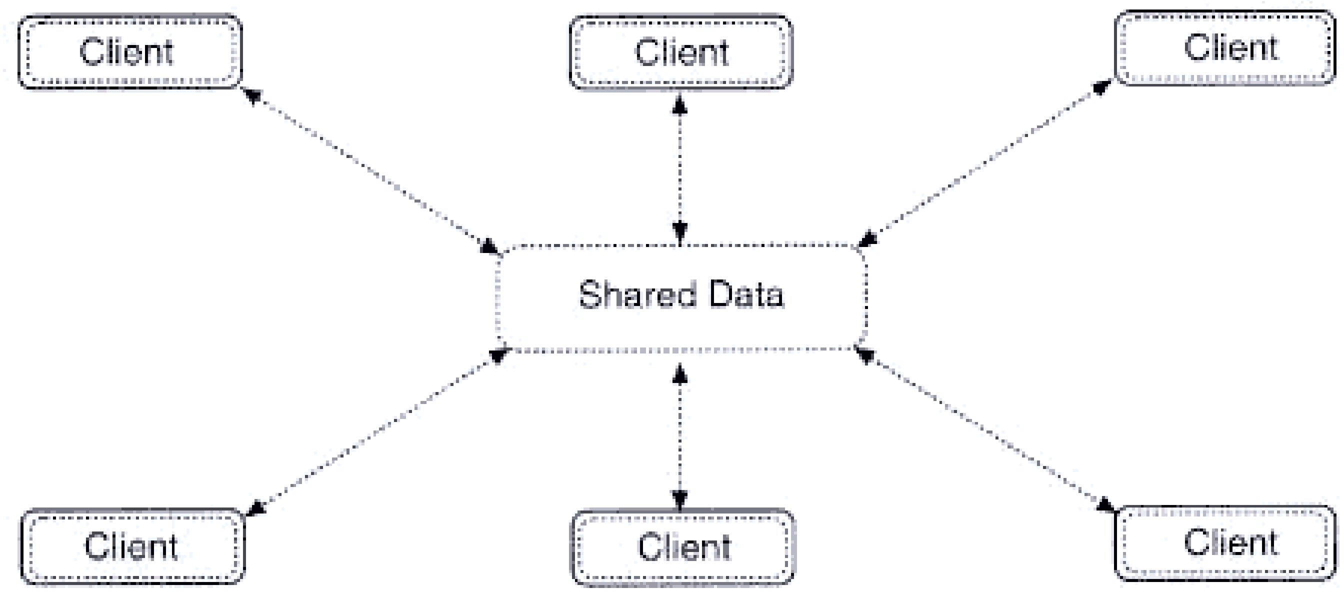
- Independent Components
  - Event Systems
  - Communicating Processes
- Data Flow
  - Batch sequential
  - Pipes and filters
- Data-Centered
  - Repository
  - Blackboard

Classification

- Virtual Machine
  - Interpreter
  - Rule-based system
- Call and Return
  - Layered
  - Object-oriented
  - Main program and subroutine
- Heterogeneous styles
  - different styles mixed at different levels

Data-centered

Repository and **Blackboard**

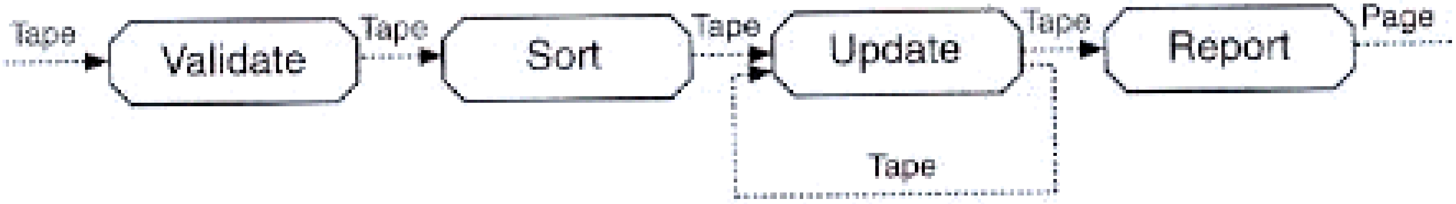


QA: D

Exercise: Name some examples of systems

Data-flow

Batch-sequential and **Pipes-and-filters**

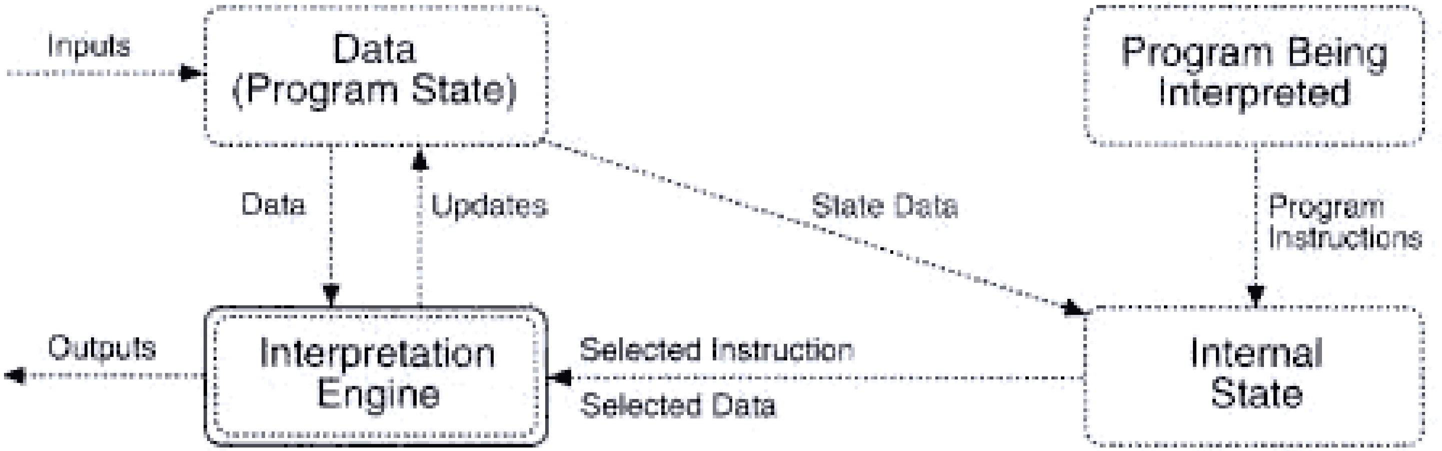


QA: Modifiability, Reusability

Exercise: Name some examples of systems

Virtual machine

Interpreter　　Rule-based systems



QA: Portability

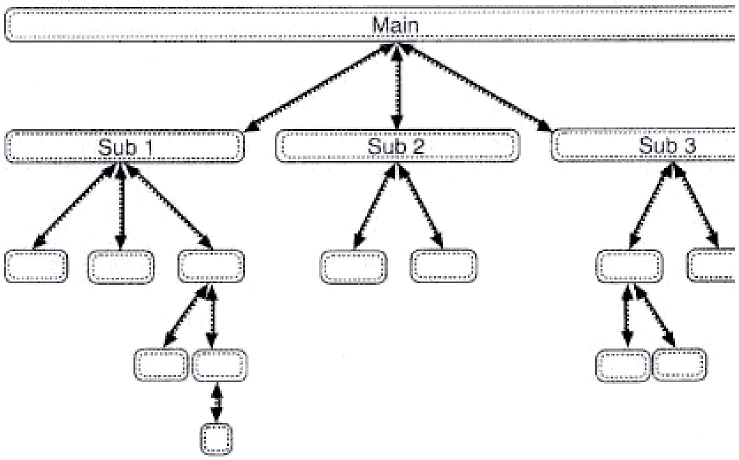
Exercise: Name some examples of systems

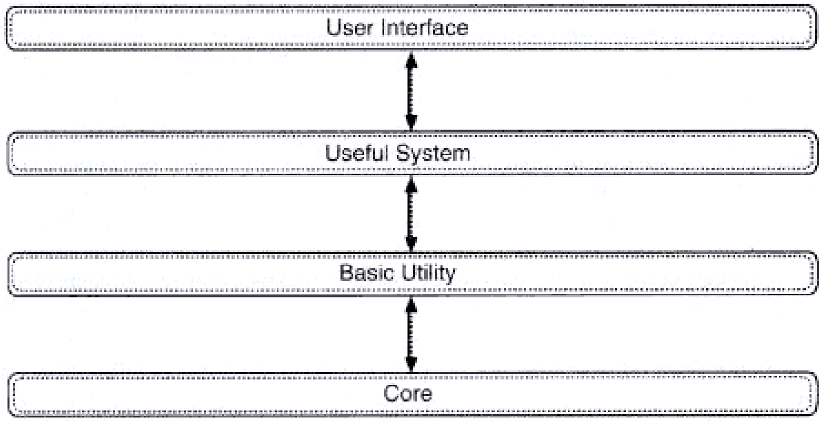
Call-and-Return

MPS style

[OO style]

Layered style





QA: Modifiability, Scalability (Layers:Portability)

Independent components

Communicating processes

- client-server is a prominent case

Event systems

- publish-subscribe systems
- message/channel based systems

QA: Modifiability (decouple sender and receiver)

Heterogeneous styles

Most large systems use several styles in a mix

The categories are not disjoint

Ex.: CORBA-based client-server

- Object-oriented call-and-return
- Layered
- Independent components

Summary

Architectural styles/patterns are proven templates for organizing components and connectors to achieve certain QA.

Can be classified

- Data-flow
- Data-centred
- Communicating processes
- Call and return

Most real architectures are mixes of styles.

Architectural Patterns

Same wine on new bottles?

Christopher Alexander: Pattern

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Christopher Alexander worked on planning towns and buildings, but the definition works just as well for object-oriented patterns.

In software the solution is expressed in terms of objects, roles, interfaces, and collaboration patterns instead of windows, doors and walls, but the contents of a patter is always:

A solution to a problem in a context

‘Alcoves’: One of Alexander’s patterns

*179. Alcoves ***

... many large rooms are not complete unless they have smaller rooms and alcoves opening off them. ♦ ♦ ♦

No homogeneous room, or homogeneous height, can serve a group of people well. To give a group a chance to be together, as a group, a room must also give them the chance to be alone, in one’s and two’s in the same space.

This problem is felt most acutely in the common rooms of a house – the kitchen, the family room, the living room. In fact it is so critical there, that the house can drivee the family apart when it remains unsolved...

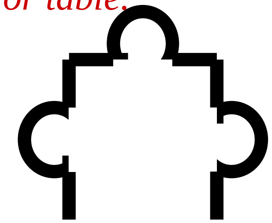
In modern life, the main function of a family is emotional; it is a source of security and love. But these qualities will only come into existence if the members of the house are physically able to be together as a family.

This is often difficult. The various members of the family come and go at different times of day; even when they are in the house, each has his own private interests...

To solve the problem, there must be some way in which the members of the family can be together, even when they are doing different things.

***Therefore**:*

Make small places at the edge of any room, usually no more than 6 feet wide and 3 to 6 feet deep and possibly much smaller. These alcoves should be large enough for two people to sit, chat, or play and sometimes large enougf to contain a desk or table.



♦ ♦ ♦

Give the alcove a ceiling which is markedly lower than the ceiling height in the main room...

(Alexander, 1977)

Branding?

Buschmann et al. (1st ed)

- Pattern-oriented software architecture

Patterns that are

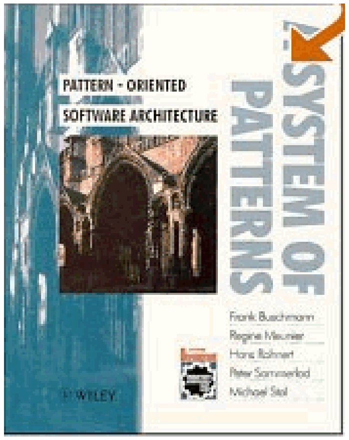
- more coarse-grained than design p.
- more specific focus than design p.

Examples

- Model-view-controller, Blackboard, Broker, Forwarder/Receiver, ...

Two other volumes

- concurrency, networking, resource management



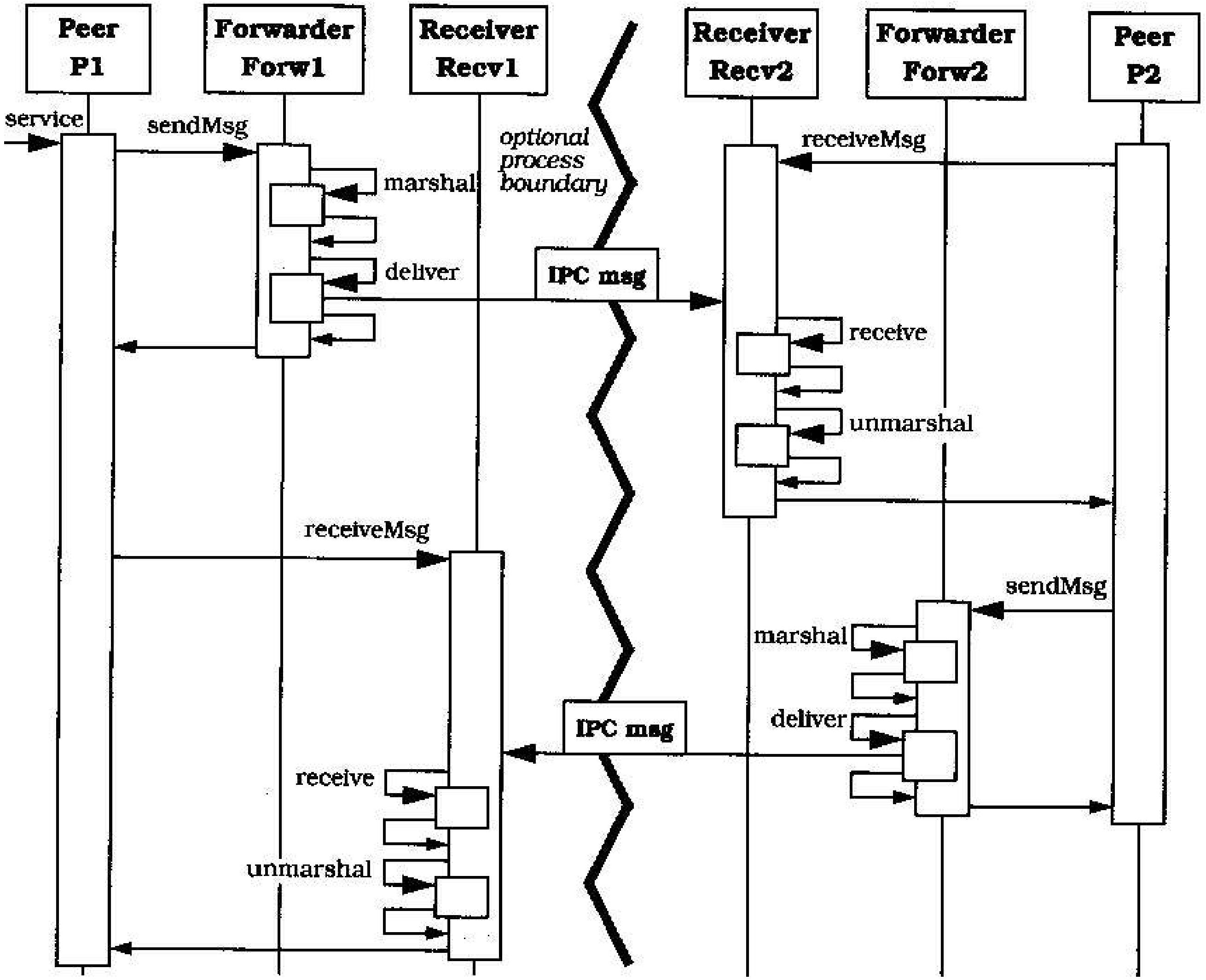
Forwarder/Receiver

Forwarder/Receiver

- **decouples Inter Process Communication**

- + portability, modifiability wrt. network IPC
- + marshalling/unmarshalling
- - modifiability wrt. re-configurations

Forwarder/Receiver



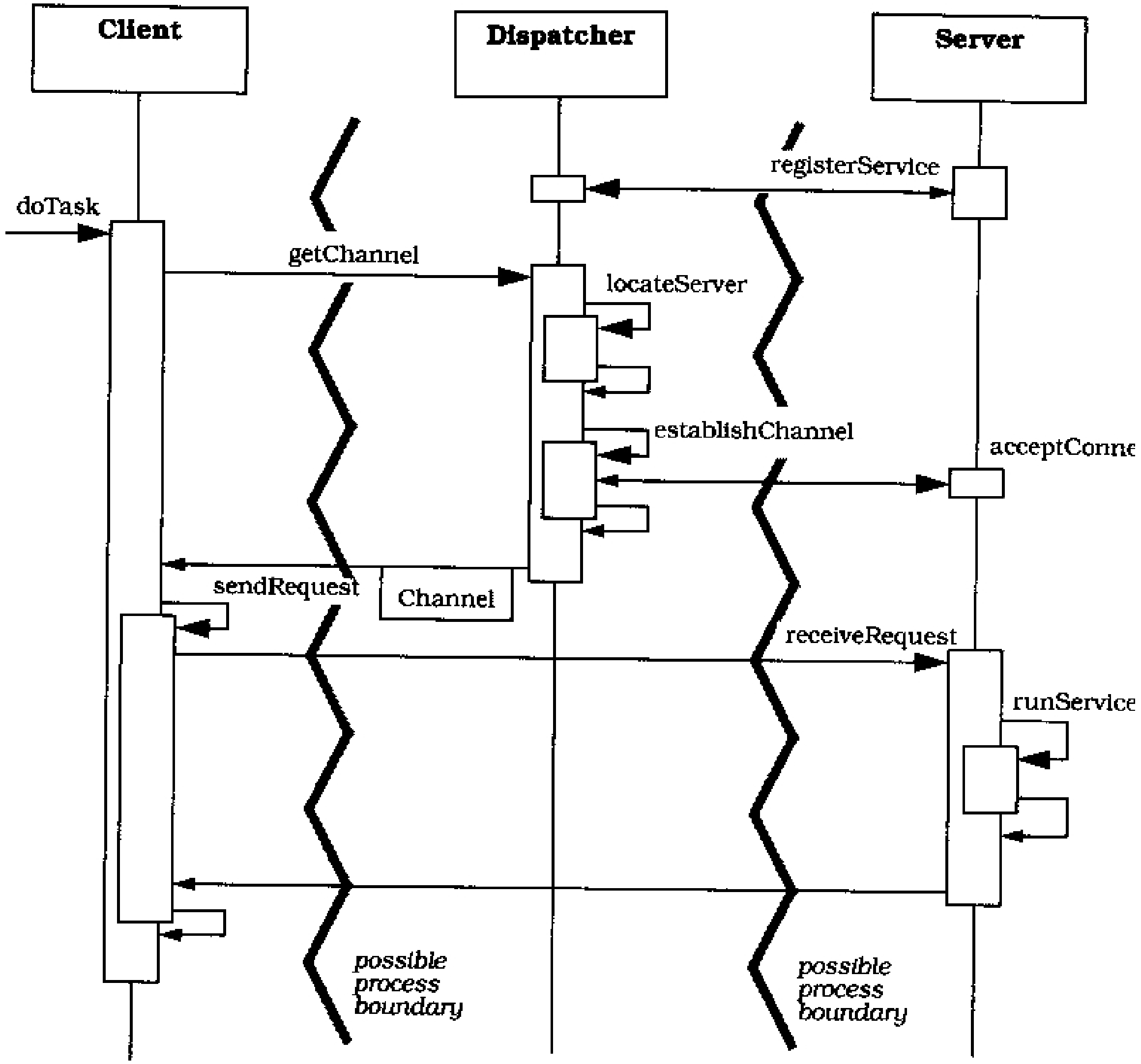
Client/Dispatcher/Server

Client/Dispatcher/Server

- **provide location transparency**

- **+** modifiability wrt. location
- **-** performance
- **-** does not encapsulate IPC
- - no marshalling

Client/Dispatcher/Server



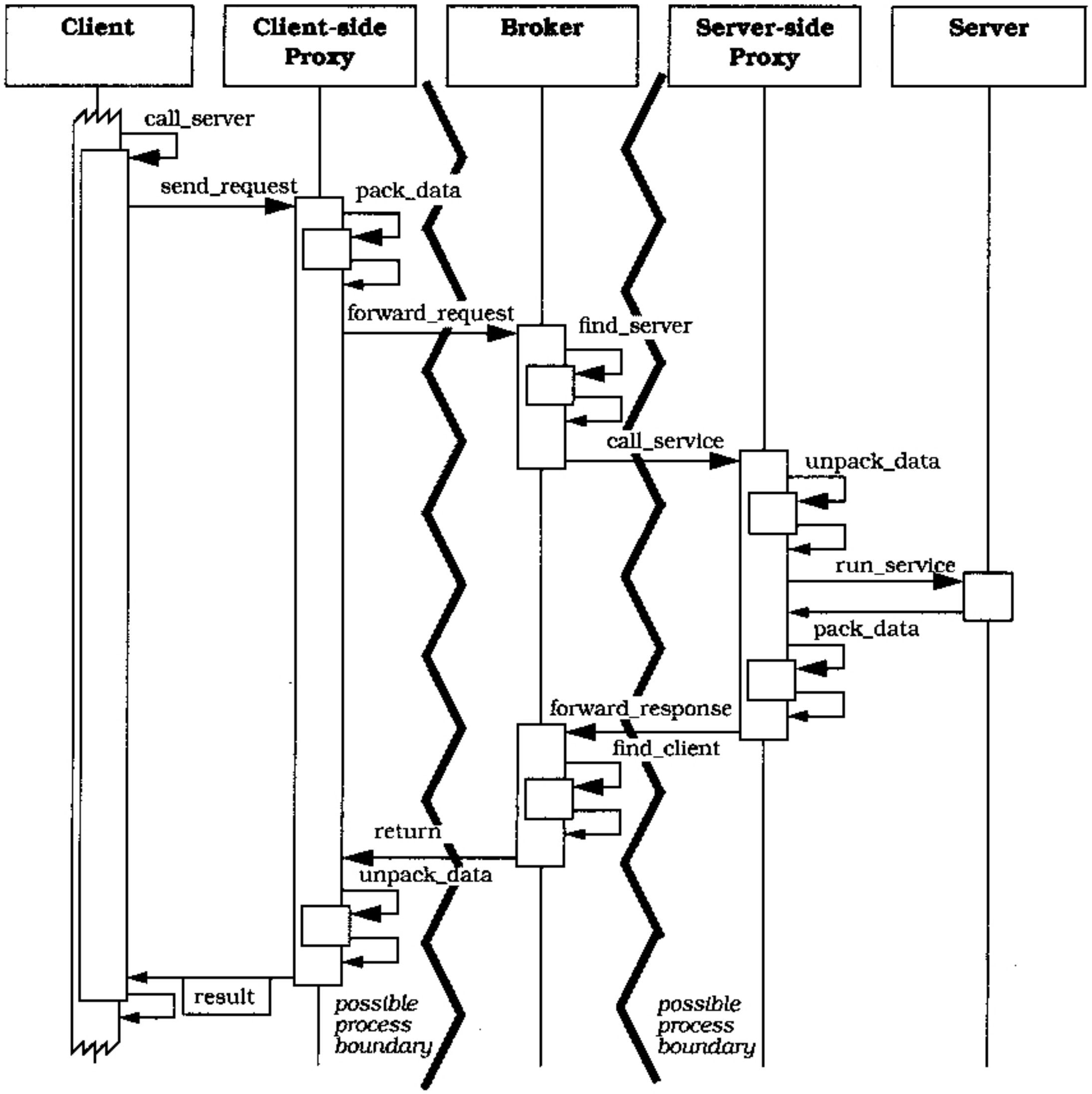
Broker (Java RMI, .NET Remoting)

Recipe:

- Take one forwarder/receiver and combine it with a client/dispatcher/server
- Add rules for marshalling, a request/reply protocol, definition of identity, and error handling
- Fry for half a minute in an IDL-to-code generator
- Spice it up with some directory service

- Serve it running ☺

Broker


Client
Client-side Proxy
Broker
Server-side Proxy
Server
call_server
send_request
pack_data
forward_request
find_server
call_service
unpack_data
run_service
pack_data
forward_response
find_client
return
unpack_data
result
possible process boundary
possible process boundary

Discussion

Relate to

- CORBA
- TS-05

Summary

As Broker shows, architectural patterns

- may be much more complex than design patterns
  - involving a lot of sub-patterns, tools, protocols, constraints
- deal with problems a higher level of abstraction – more “architectural”
- much more restricted in its usage compared to design patterns

Tactics

Surgical means for getting a quality

Architectural Tactics

Tactic

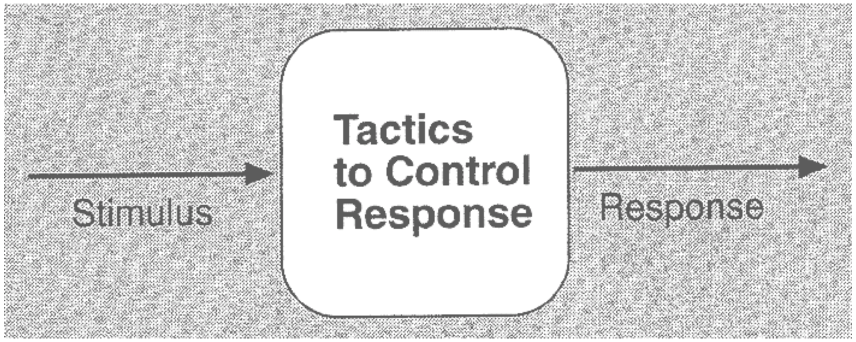
- A design decision that influences the control of a quality attribute response
- E.g., *Heartbeat* to control availability

Architectural strategy

- Collection of tactics

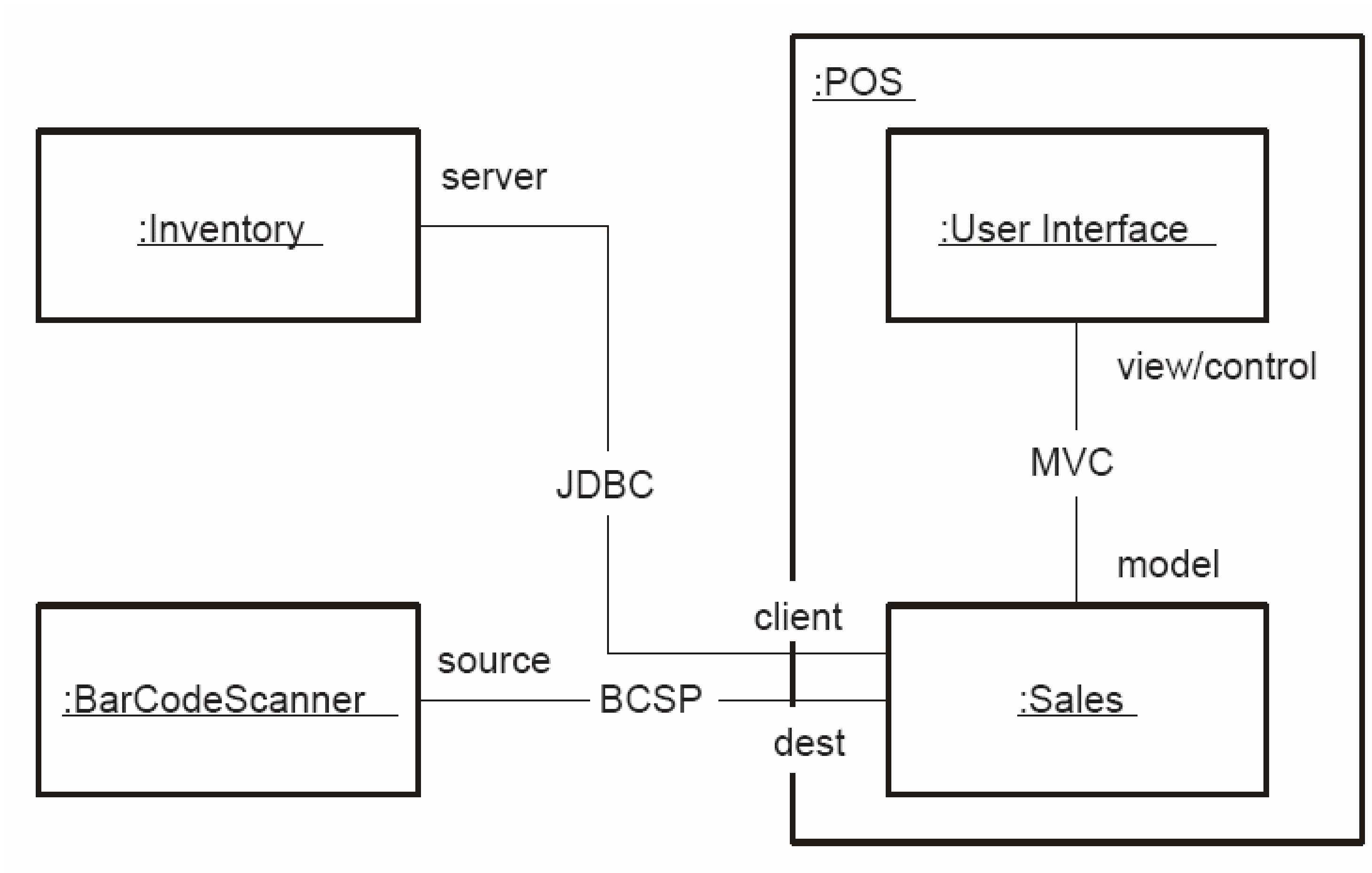
Characteristics

- Capture what architects do in practice
- Tactics may refine other tactics
- Tactics may influence more than one quality attribute
  - Since quality attributes are interdependent

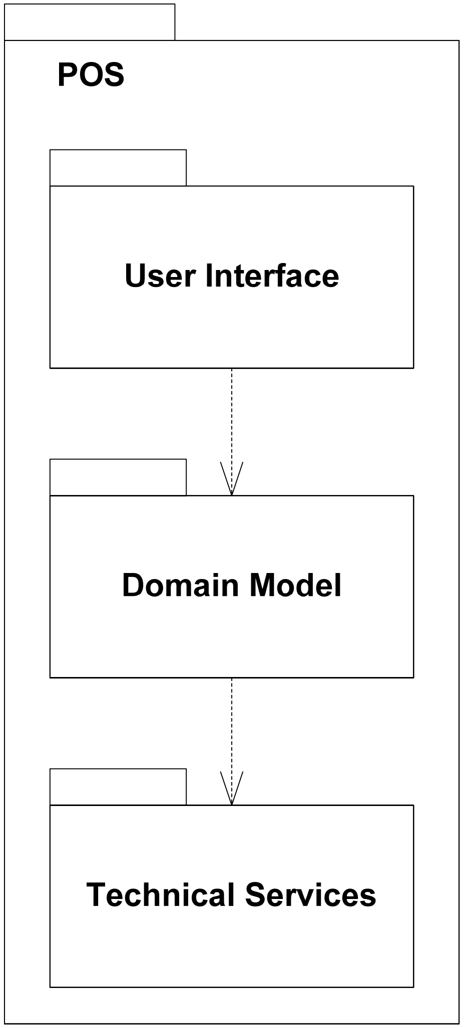


AARHUS UNIVERSITET

POS Revisited – Component and Connector View

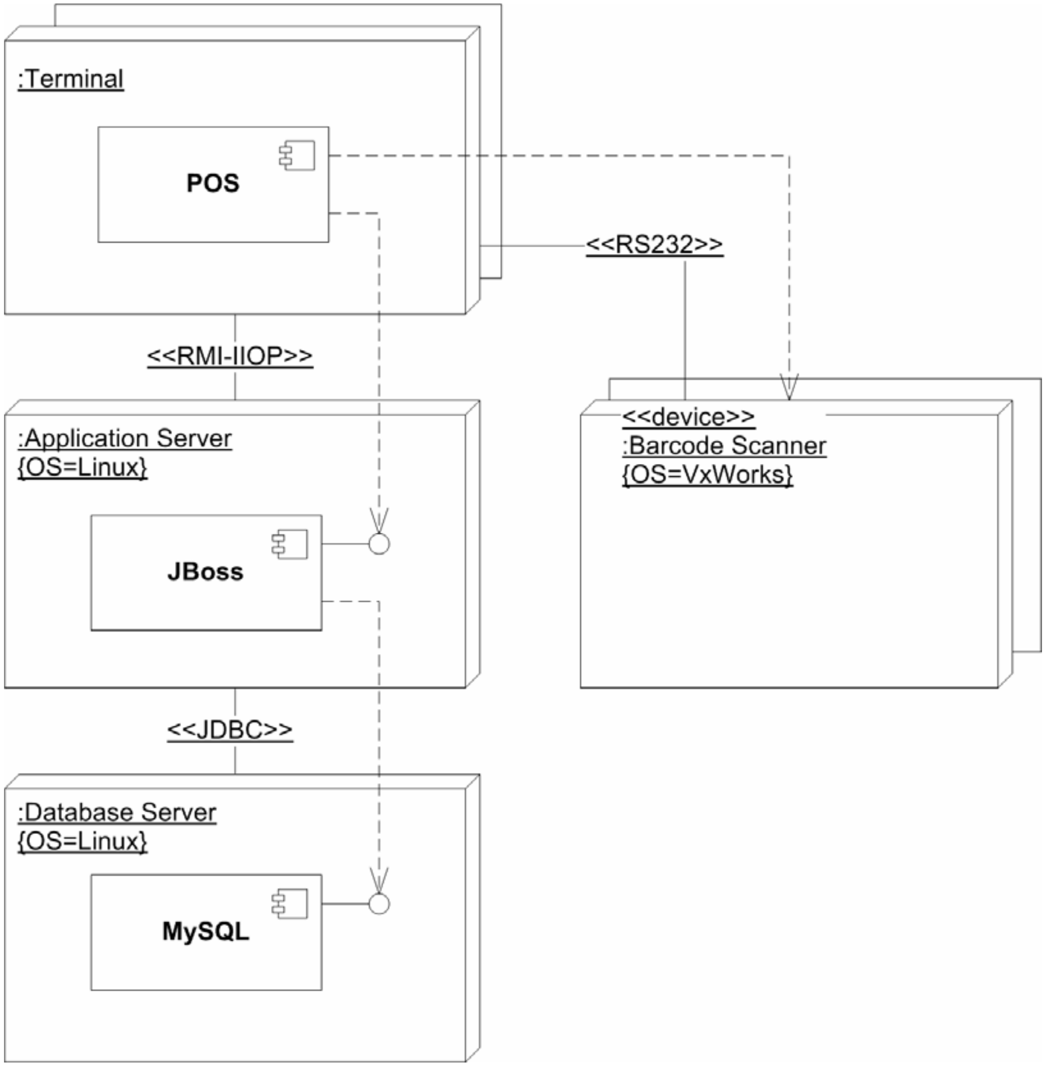


POS Revisited – Module View



(Revised compared to [Christensen et al, 2007])

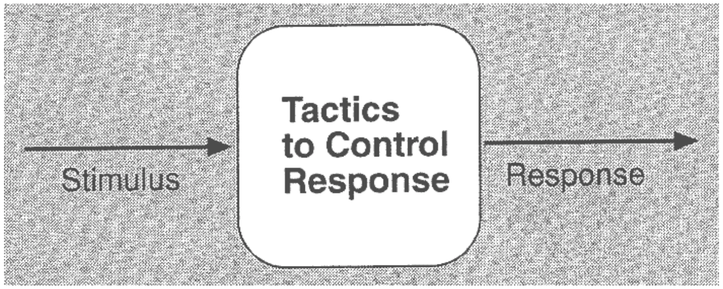
POS Revisited – Allocation View



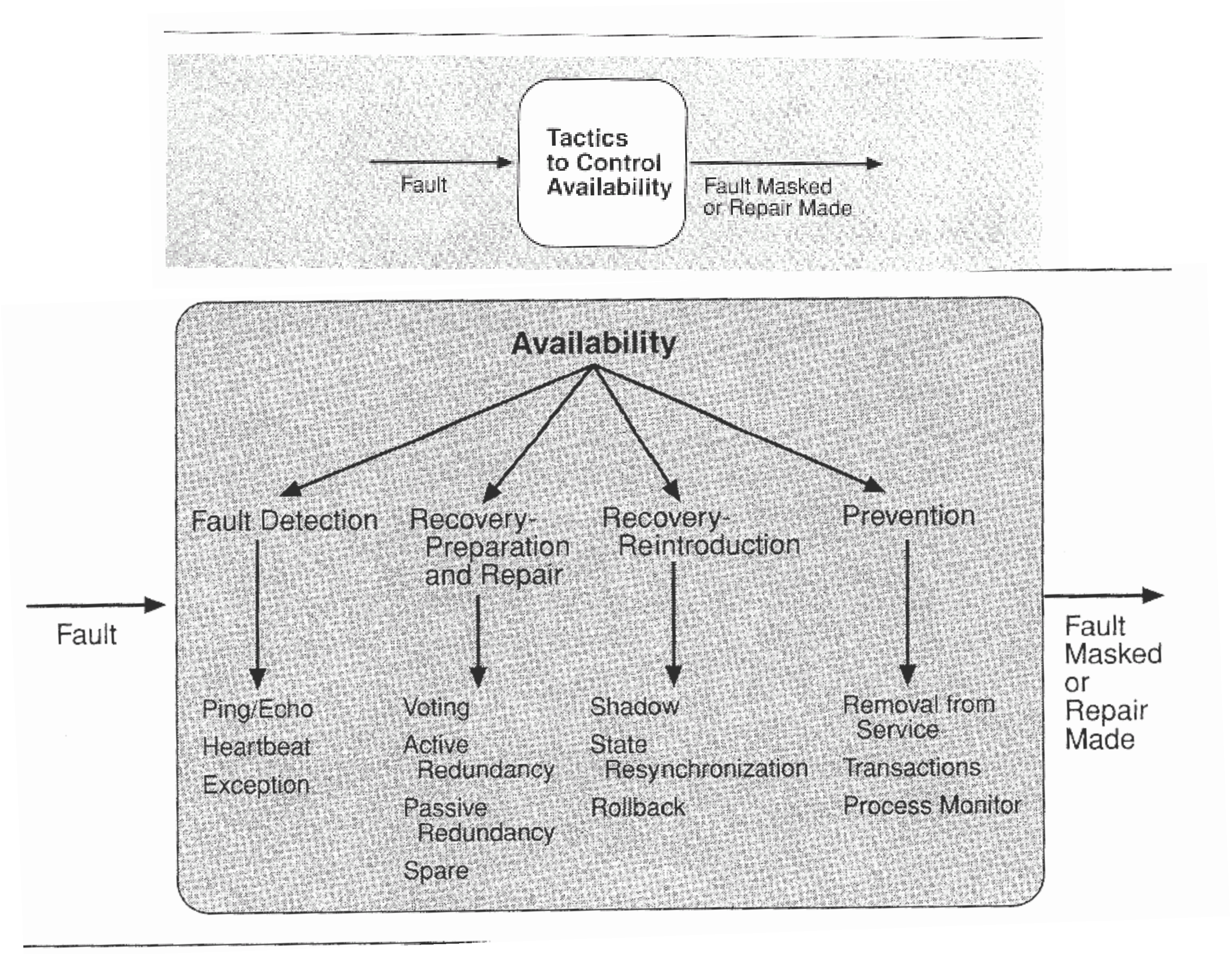
Categories of Tactics

Classified according to (main) quality attribute concern

- Availability
- Modifiability
- Performance
- Security
- Testability
- Usability



Availability Tactics (1)



Availability Tactics (2)

Fault detection

- Ping/echo
  - One component pings
  - Expects response within predefined time
- Heartbeat (dead man timer)
  - One component emits heartbeat message periodically
  - Other components listen for it
- Exceptions
  - Raise exception when fault class is encountered
  - Omission, crash, timing, response fault

Fault recovery – repair

- Voting
  - Redundant processes and processors
  - Voter process check responses – fail if deviant
- Active redundancy (hot restart)
  - Maintain redundant, parallel components
  - Only use one response
- Passive redundancy (warm restart)
  - Primary component responds, informs standbys of updates to make
  - Resume standby if primary fails
- Spare
  - Standby computing platform
  - Boot and initialize state when needed

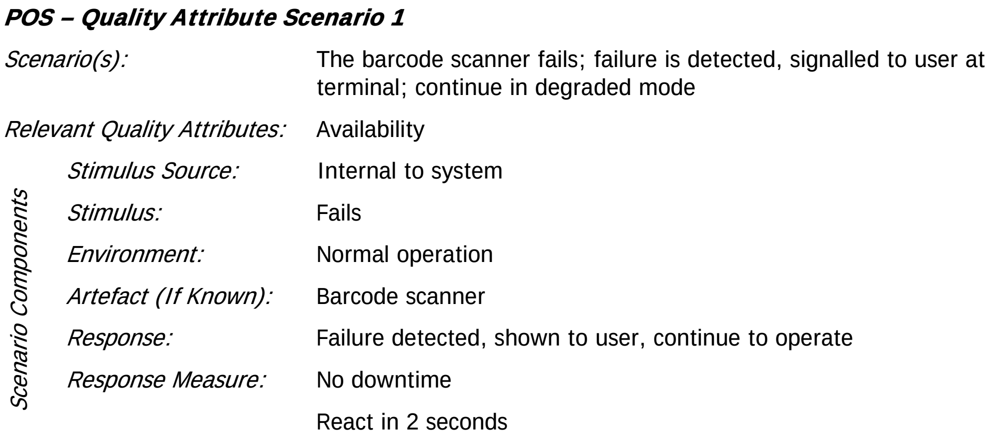
Fault recovery – reintroduction

- Shadow operation
  - Previously failed component runs in “shadow mode”
  - Restore when sure that it works
- State resynchronization
  - Redundancy requires restoring after downtime
- Checkpoint/rollback
  - Create checkpoint recording consistent state at points in time
  - Rollback to previous checkpoint if inconsistent state detected

Fault prevention

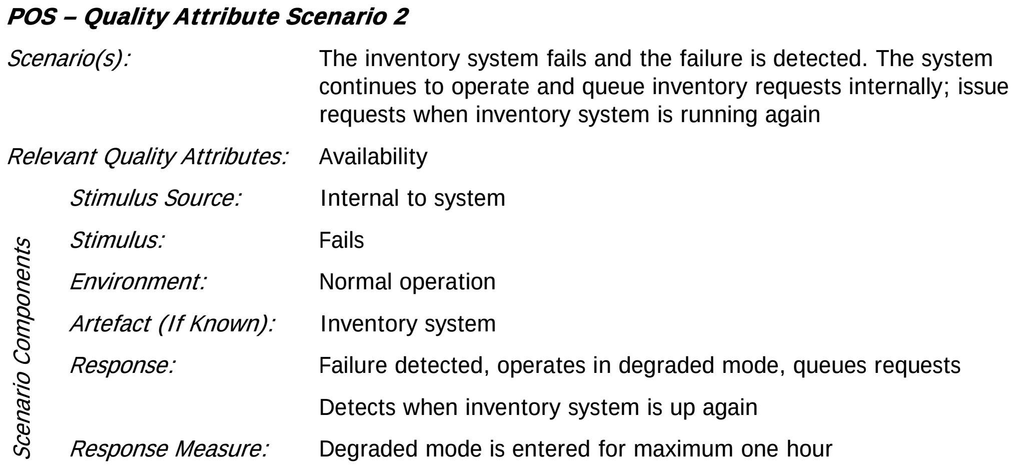
- Removal from service
  - (Periodically) remove component to prevent anticipated failure
- Transactions
  - Bundling sequential steps
  - Undo all if necessary

POS Availability Scenarios



POS – Quality Attribute Scenario 1

Scenario(s):		The barcode scanner fails; failure is detected, signalled to user at terminal; continue in degraded mode
Relevant Quality Attributes:		Availability
Scenario Components	*Stimulus Source:*	Internal to system
	Stimulus:	Fails
	Environment:	Normal operation
	Artefact (If Known):	Barcode scanner
	Response:	Failure detected, shown to user, continue to operate
	Response Measure:	No downtime
		React in 2 seconds



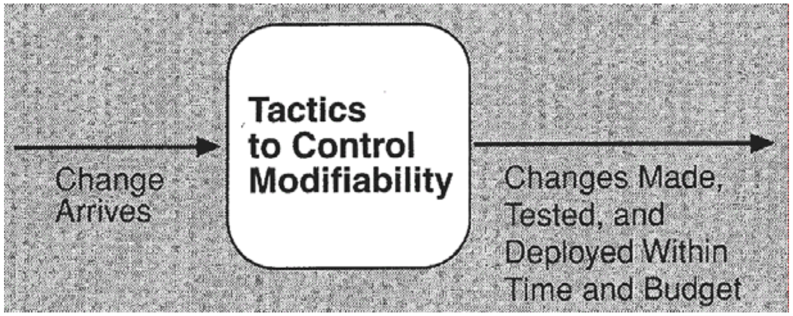
POS – Quality Attribute Scenario 2

Scenario(s):		The inventory system fails and the failure is detected. The system continues to operate and queue inventory requests internally; issue requests when inventory system is running again
Relevant Quality Attributes:		Availability
Scenario Components	*Stimulus Source:*	Internal to system
	Stimulus:	Fails
	Environment:	Normal operation
	Artefact (If Known):	Inventory system
	Response:	Failure detected, operates in degraded mode, queues requests
		Detects when inventory system is up again
	Response Measure:	Degraded mode is entered for maximum one hour

Exercise

- – Which tactics can be used to handle the scenarios?
- – Are other tactics relevant to POS?

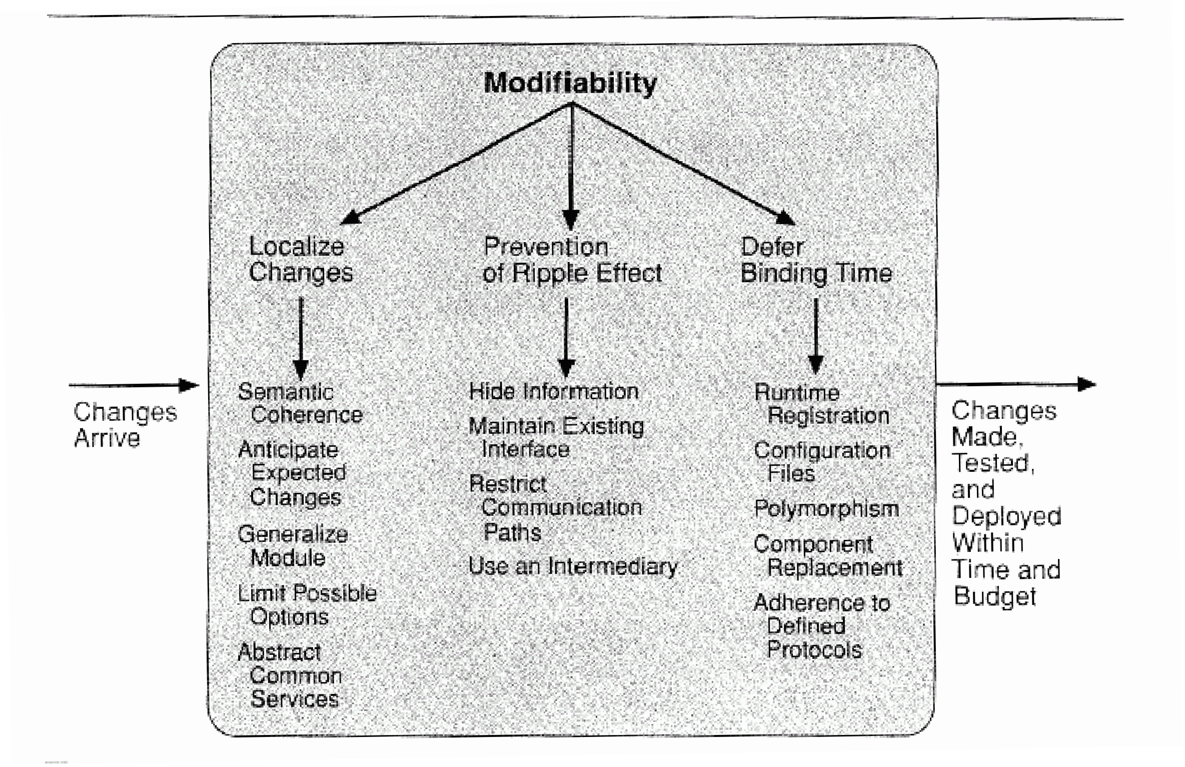
Modifiability Tactics (1)



Change Arrives

Tactics to Control Modifiability

Changes Made, Tested, and Deployed Within Time and Budget



Changes Arrive

Modifiability

Localize Changes

- Semantic Coherence
- Anticipate Expected Changes
- Generalize Module
- Limit Possible Options
- Abstract Common Services

Prevention of Ripple Effect

- Hide Information
- Maintain Existing Interface
- Restrict Communication Paths
- Use an Intermediary

Defer Binding Time

- Runtime Registration
- Configuration Files
- Polymorphism
- Component Replacement
- Adherence to Defined Protocols

Changes Made, Tested, and Deployed Within Time and Budget

Modifiability Tactics (2)

Assumption

- *Restricting modifications to small set of module will reduce cost of change*

Localize changes

- Semantic coherence
  - Ensure responsibilities of a module are coherent
  - Low coupling + high coherence + measured against scenarios of change
- Anticipate expected changes
  - Make decomposition so that considered changes affect minimal number of modules
  - Based on assumptions of what changes will be
- Generalize module
  - Make module compute broader range of functions
  - E.g., constants -> input parameters
- Limit possible options
  - Reduce options for modifications

Prevention of ripple effect

- Hide information
  - Decompose responsibilities
  - Choose which to make public, hide others
- Maintain existing interface
  - Mask variations
- Restricts communication paths
  - Restrict the number of module with which a component shares data
- Use an intermediary
  - Create module handling dependencies between components (e.g., Adapter)

Defer binding time

- Runtime registration
- Configuration files
- Polymorphism
- Component replacement
- Adherence to defined protocols

POS Modifiability Scenario

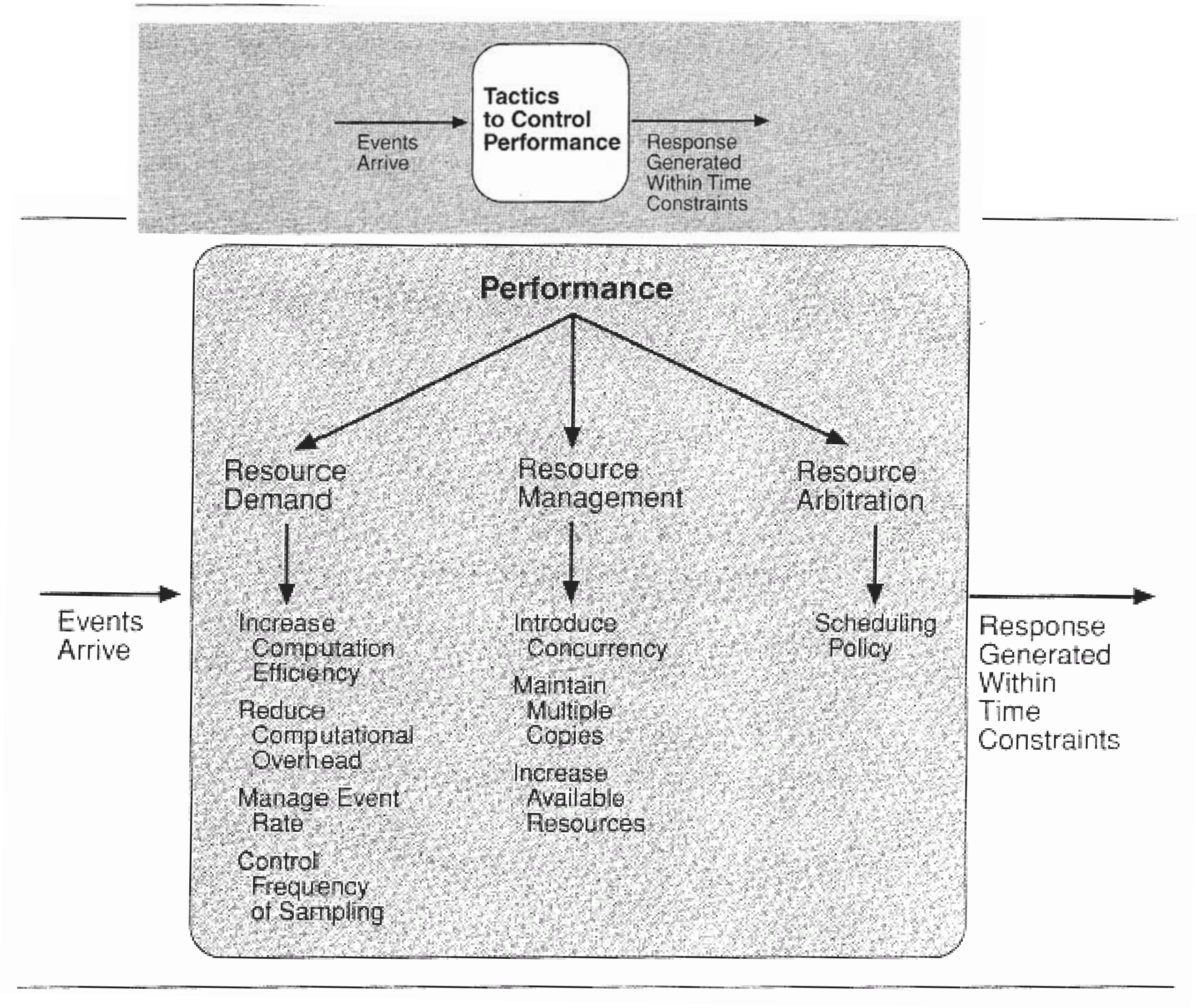
POS – Quality Attribute Scenario 3

Scenario(s):		The POS system should be extended to handle “supermarket” domains as well as “small shop” domains
Relevant Quality Attributes:		Modifiability
Scenario Components	*Stimulus Source:*	Developers
	Stimulus:	Wants to change domain of POS
	Environment:	Development time
	Artefact (If Known):	POS system
	Response:	Domain is changed
	Response Measure:	Cost of change is “reasonable”

Exercise

- Which tactics can be used to handle the scenario?
- Are other tactics relevant to POS?

Performance Tactics (1)

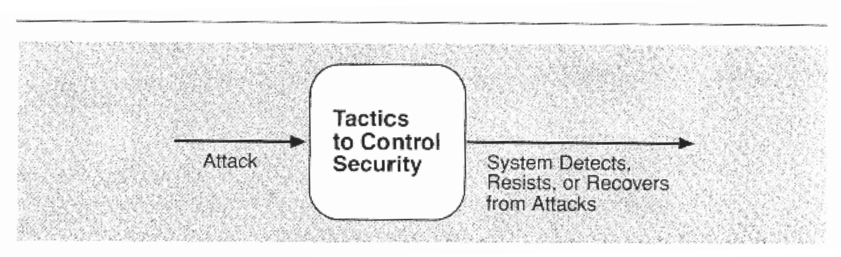


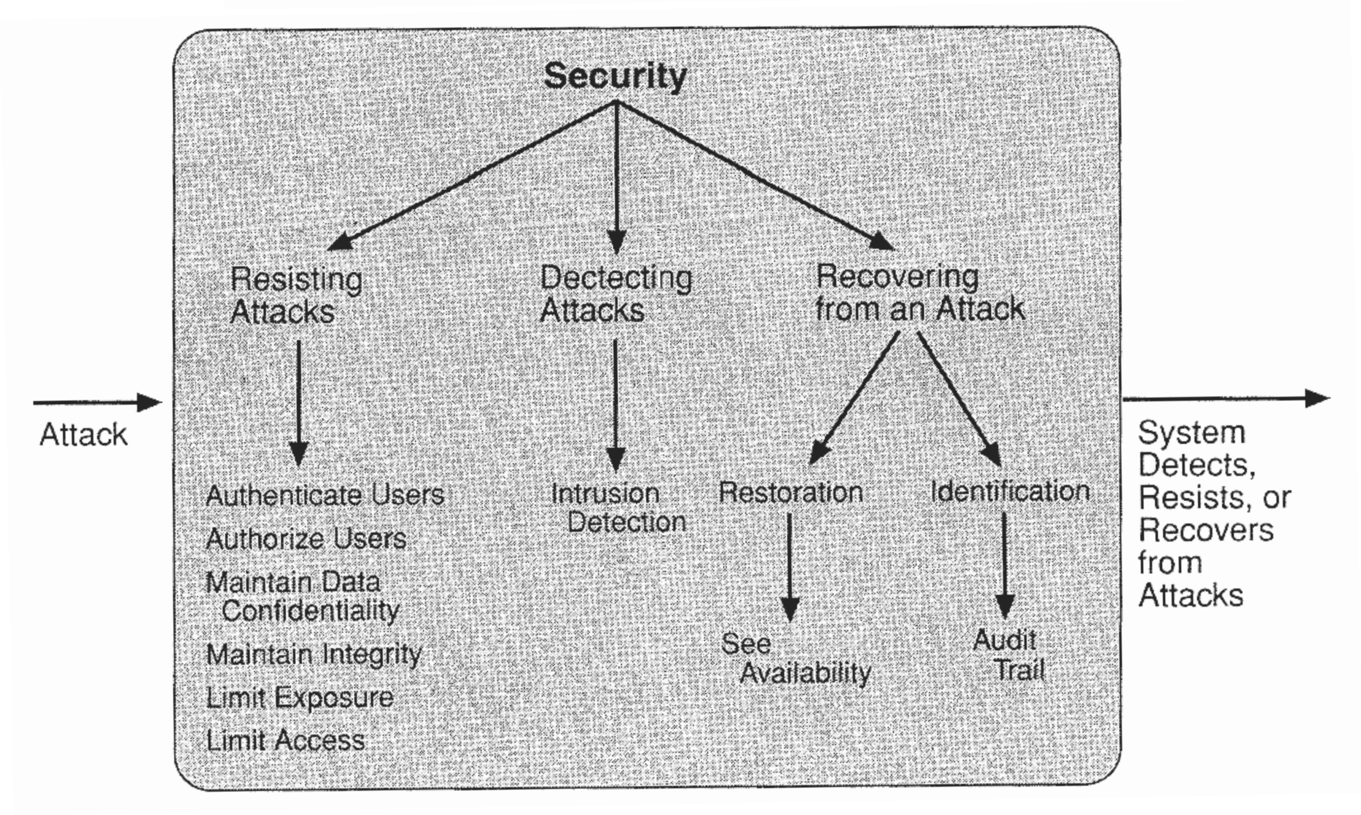
Performance Tactics (2)

POS – Quality Attribute Scenario 4

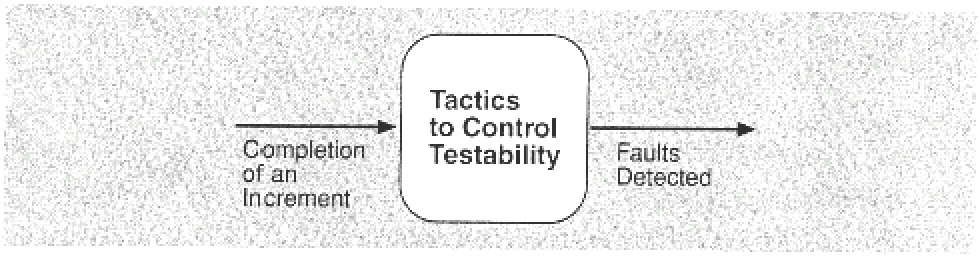
Scenario(s):		The POS system scans a new item, item is looked up, total price updated within two seconds
Relevant Quality Attributes:		Performance
Scenario Components	*Stimulus Source:*	End user
	Stimulus:	Scan item, fixed time between events for limited time period
	Environment:	Development time
	Artefact (If Known):	POS system
	Response:	Item is looked up, total price updated
	Response Measure:	Within two seconds

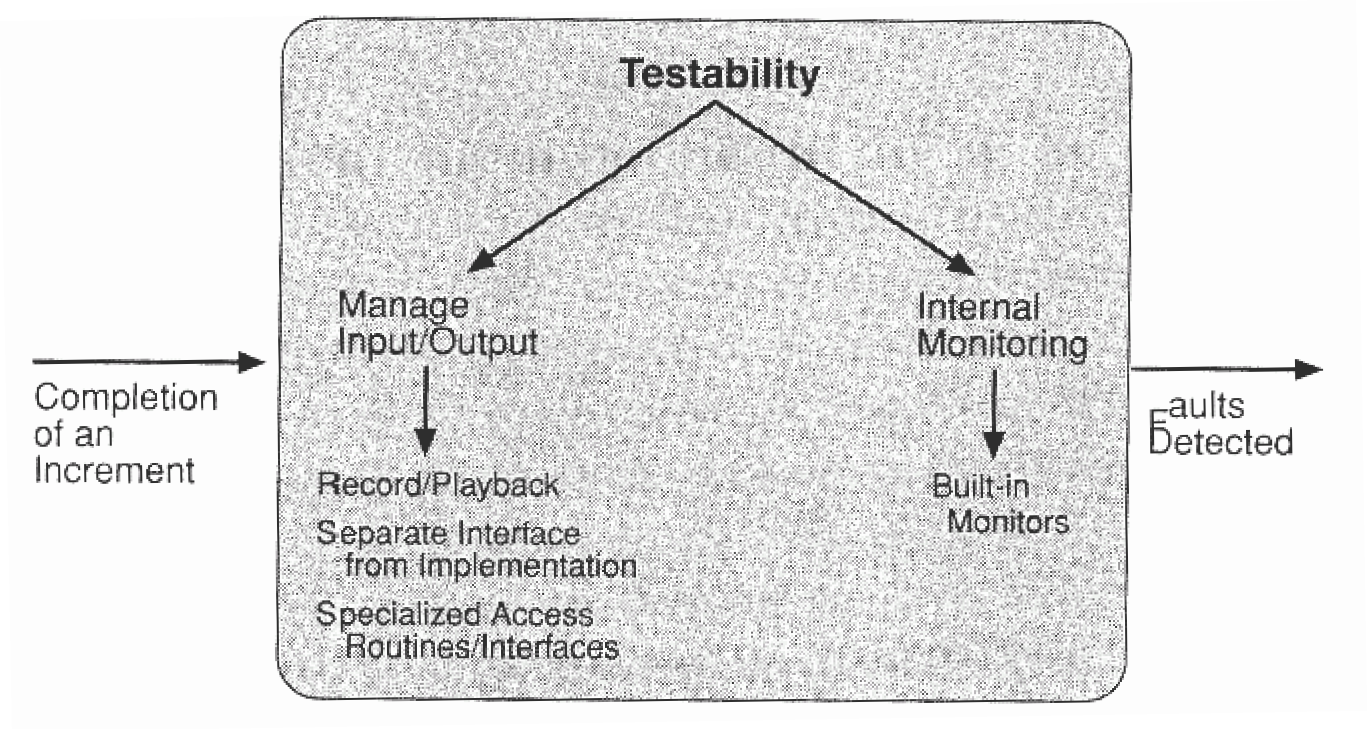
Security Tactics


Attack
Tactics to Control Security
System Detects, Resists, or Recovers from Attacks

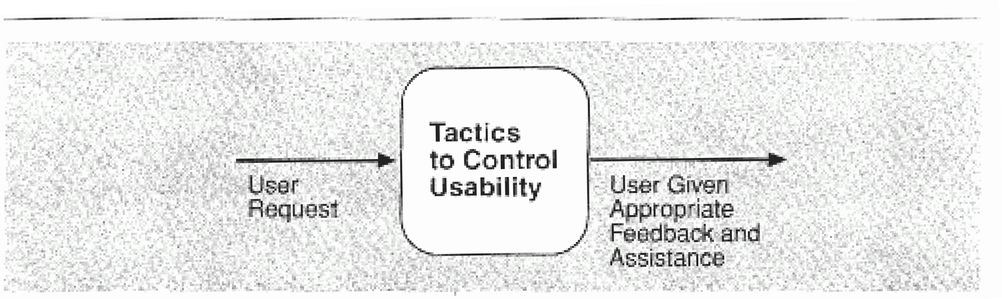


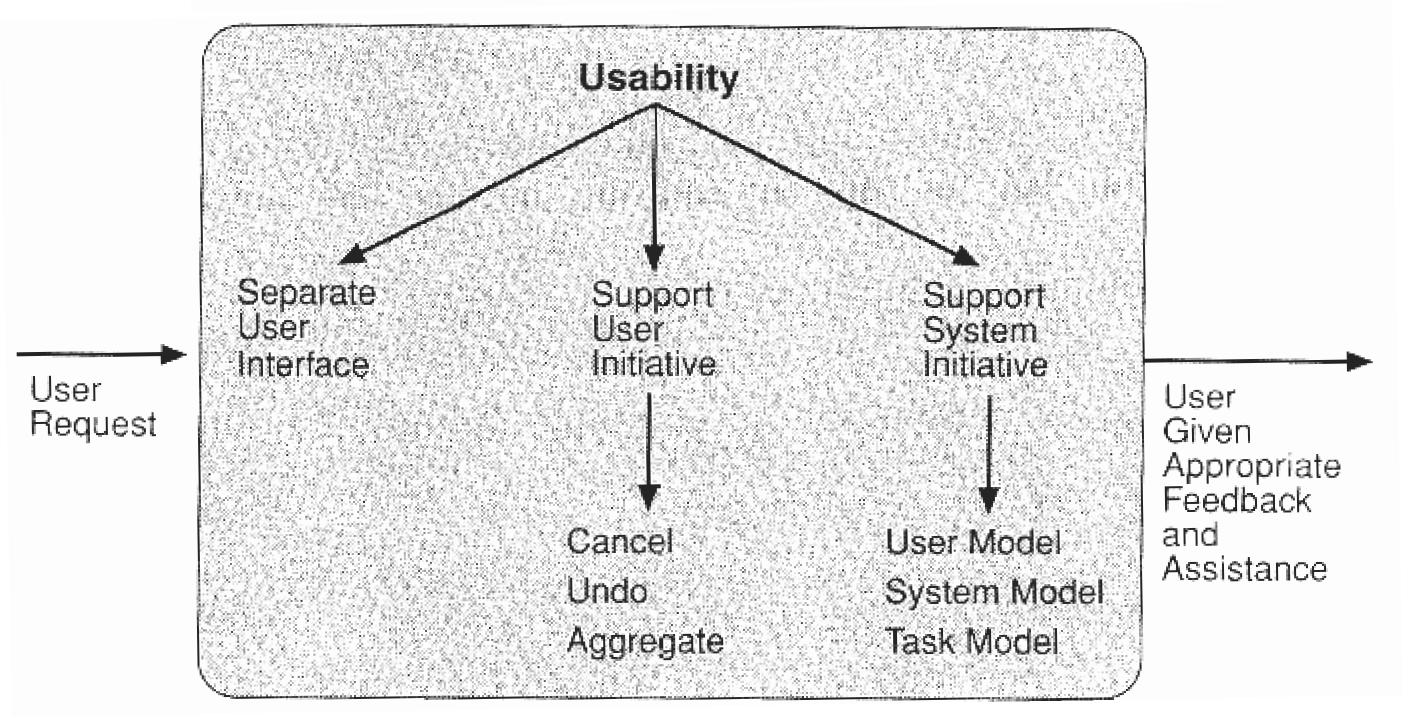
Testability Tactics





Usability Tactics


User Request
Tactics to Control Usability
User Given Appropriate Feedback and Assistance



Discussion

Tactics help make quality attribute decisions

- Does it make sense to divide tactics according to quality attributes – cf. interdependence?

Do tactics make sense regardless of domain?

Are the tactics really just design ideas?

- Cf. patterns…