Architectural synthesis

Week 3

IT UNIVERSITY OF COPENHAGEN

Objectives

To understand in which way architectural synthesis is not a rational activity

To provide techniques for deciding the overall structure of systems

To provide techniques for deciding the detailed structure of systems

Topics covered

The synthesis process

Architectural styles

Architectural tactics

Today's reading

1. **A Rational Design Process: How and Why to Fake It**. David Lorge Parnas and Paul C. Clements (1987) *IEEE Transactions on Software Engineering*. SE-12(2), pp 251-256
2. **Architectural Patterns Revisited – A Pattern Language**. Avgeriou & Zdun. In Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), pp 1--39, 2005
3. **Software Architecture in Practice**, 4rd Edition. Bass, Clements, Kazman. Addison-Wesley, 2021.
    - Section 4.2 (availability tactics)
    - Section 8.2 (modifiability tactics)
    - Section 9.2 (performance tactics)
    - Section 11.2 (security tactics)

1. The Synthesis Process

Architectural Synthesis (AS): proposing a collection of architecture solutions to address the ASRs that are identified during AA [15]. This activity essentially links the problem to the solution space.

Application of knowledge-based approaches in software architecture: A systematic mapping study

Zengyang Li, ... Paris Avgeriou, in Information and Software Technology, 2013

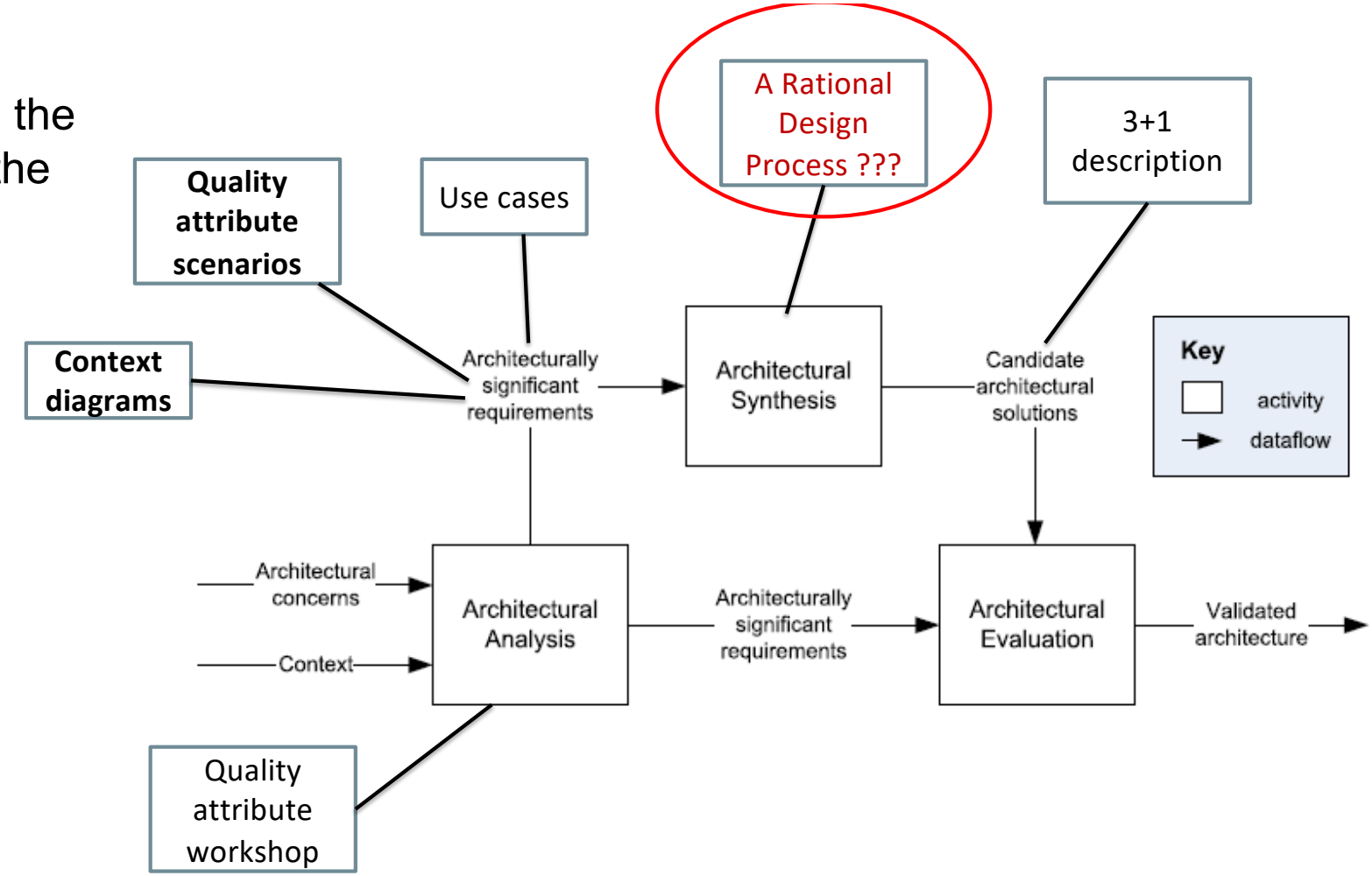
Why is synthesis important?

- Proposed architecture **defines the framework in which the rest of the development will fit in**
  - Think monolith vs. microservices

- **Forces you to think deep upfront**
  - Even if often **it must be iterated** (just as in real architecture)

How?

Could we ”**derive**” the architecture from the requirements?



2. A Rational Design Process

Could we ”derive” the architecture from the requirements?

A Rational Design Process: How and Why to Fake It. David Lorge Parnas and Paul C. Clements (1987) *IEEE Transactions on Software Engineering.* SE-12(2), pp 251-256

A **rational design process**: **why** you can't have it

Stakeholders do not necessarily know what they want or are unable to tell all they know

even if… many **details only become known during implementation**

even if… projects are subject to **change for external reasons**

even if… **people make mistakes**

even if… **the conflict of qualities**

(Parnas and Clements, 1986)

The Conflict of Qualities

Many system qualities are in direct conflict – they must be balanced!
e.g. …

Modifiability vs. Performance

- ???

Cost vs. Reusability

- ???

Security vs. Performance / Usability

The Conflict of Qualities

Many system qualities are in direct conflict – they must be balanced!
e.g. …

Modifiability vs. performance

- E.g. Many delegations costs time (and memory)

Cost vs. reusability

- E.g. Highly flexible software costs time, effort, and money

Security vs. Performance / Usability

A rational design process: **how** to fake it

Fake end result = appears as if a rational process was followed:

Requirements are clear,
design is well-described,
it is argued that design meets requirements

We are **interested in the end result, not the steps going into producing it**

(As in mathematics: a proof of a theorem does not reflect the discovery of the proof

Note: **This is the same in your thesis work or other similar research reports**

A fake synthesis process

1) **Choose architectural style(s)** according to quality requirements

   Styles = topic of today's lecture

2) **Create overall, tentative structure**

   Q1: What is the context of the system?

   Describe tentatively the other systems and actors the system interacts with

   Q2: What functionality is needed?

   Describe tentatively **components** (i.e. functionality) **& connectors**

   Describe **where** components are to be deployed

   Q3: How do you *divide the work* of implementing functionality?

   Describe tentative **modules** (e.g. packages)

The result of (2) is the set of initial architectural UML models corresponding to the 3+1 approach

A fake synthesis process

3) **Refine initial structure** through quality attribute scenarios:

For each quality attribute scenario **apply architectural tactics** for the main quality attribute

Consider applying tactics also for quality attributes for which you do not have scenarios

(The result of (3) are refined UML models)

Tactics = also topic of today's lecture

A fake synthesis process

4) Consider architectural and business qualities

Conceptual integrity

– Could you increase integrity by doing similar things in similar ways?

Correctness and completeness

- Do you cover functional requirements correctly?

Feasibility

- With the resources available, is it possible to realize the system?

Conceptual integrity is the principle that anywhere you look in your system, you can tell that the design is part of the same overall design. This includes low-level issues such as **formatting and identifier naming**, but also issues such as how modules and classes are designed, etc.

A fake synthesis process

5) Maintain an architectural backlog

- For each step, note questions and uncertainty in the backlog
- Many decisions might need to be taken when more information is available
  - Detailed design, architectural prototyping, implementation …

Sometimes it's better to delay the most important decisions… don't commit to something before it's necessary.

3. Architectural Patterns

Reusable solutions to commonly occurring problems in software architecture

Architectural Patterns Revisited – A Pattern Language. Avgeriou & Zdun. In Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), pp 1--39, 2005

Terminology

Architectural pattern defines
- types of **elements** and
- **relationships** that
- work **together** in order to solve a particular problem

Architectural pattern =(in this course) **architectural style**

Pattern = focusing on problem-solution pair.
Style = focusing on the solution; not much about context.

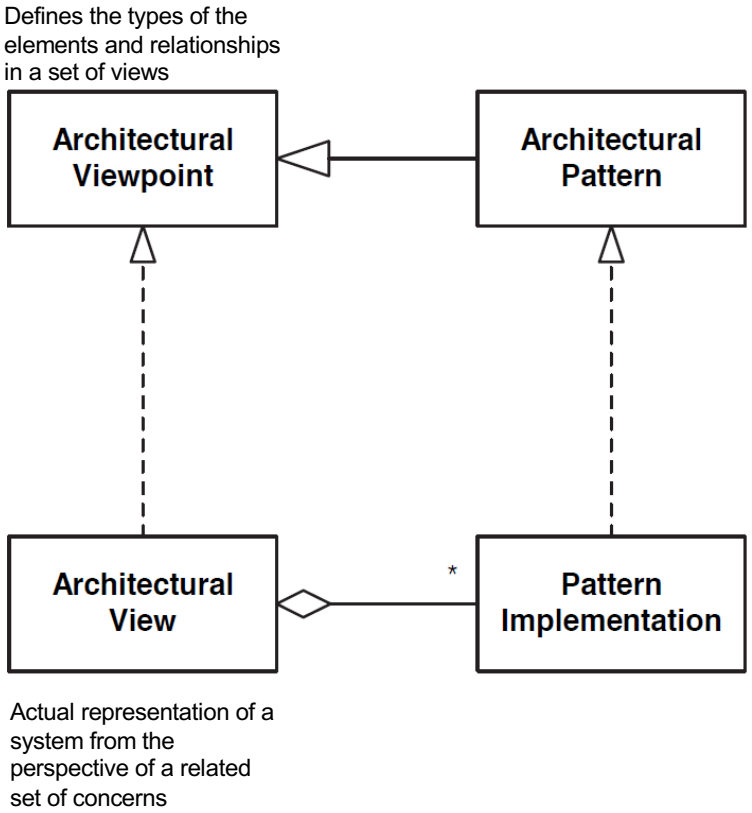
Terminology (2)

Architectural pattern defines
- types of **elements** and
- **relationships** that
- work **together** in order to solve a particular problem

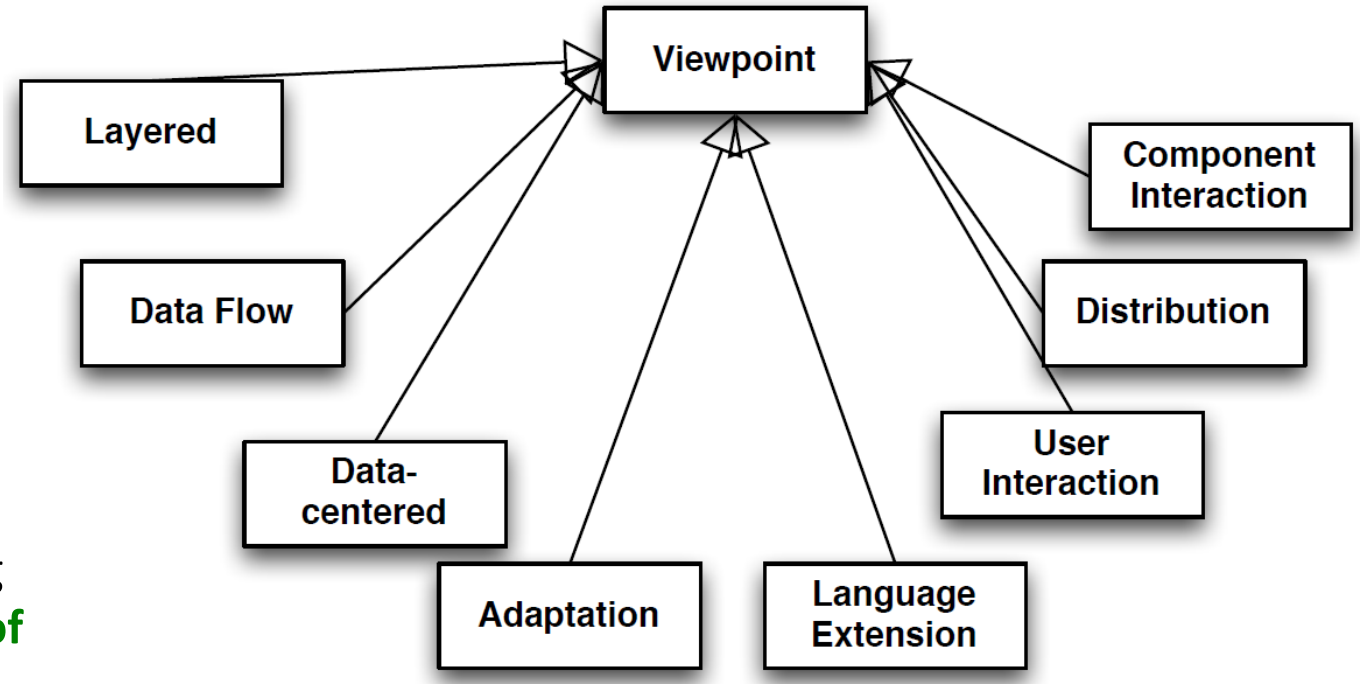
Architectural pattern =(in this course) **architectural style**

Pattern = focusing on problem-solution pair.
Style = focusing on the solution; not much about context.

A **viewpoint** focuses on relevant **concerns**
e.g. how is computation distributed over the nodes



Patterns & Viewpoints



Avgeriou and Zdun (2005) classify patterns according to ***viewpoints***, i.e., **types of perspectives on architecture**

Examples for Today
(case studies)

1: NextGen Point-Of-Sales (POS)

Record sales and handle payments

– Typically used in retail stores

Hardware

– Terminal

– Barcode scanner

Interfaces with external systems

– Inventory

– Accounting

– …

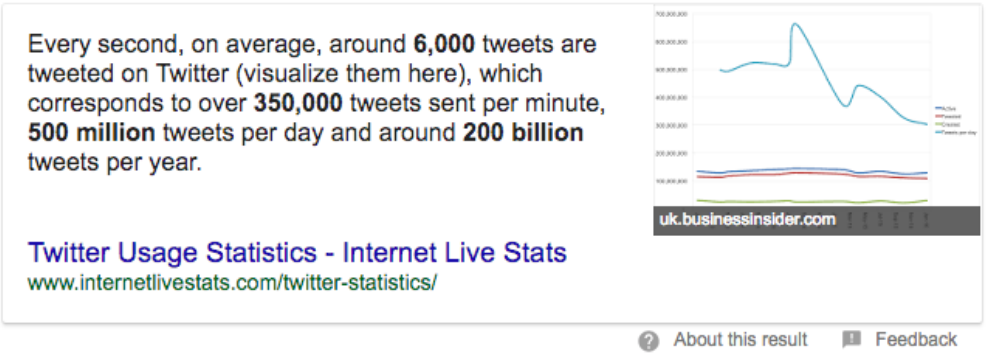
From [Larman, 2001]

2: Twitter

Multiple “user interfaces”: Web, Android, iPhone, RSS feeds, blogs, SMS

Graph of Following and Followed central

Task of architects: synthesize an architecture so that this is possible…



http://www.internetlivestats.com/one-second/#tweets-band

Viewpoint: **Component Interaction**

… system is seen as number of independent but interacting components

Concerns

- How do the independent components interact with each other?
- How are the individual components decoupled from each other?
- How are the quality attributes of modifiability and integrability supported?

Client-Server (Component Interaction)

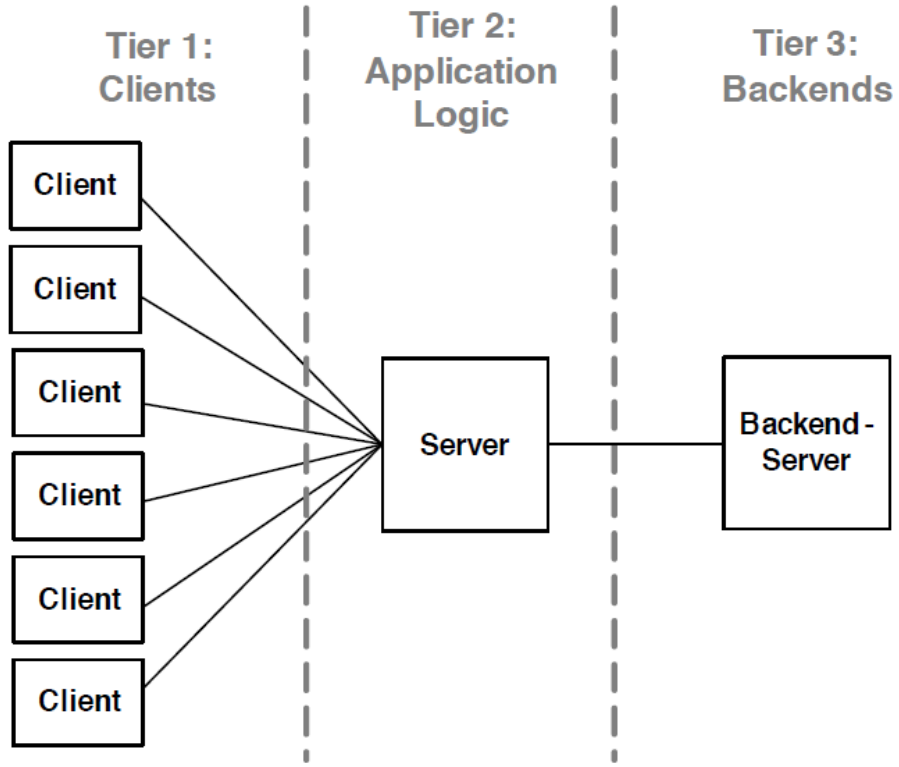
Problem: two distributed components need to communicate, one needing the service of another

Clients request information or services from the Server

The Server processes and responds to client requests

Canonical example?

- Web browser and web server
- 3-tier web applications



?

?

Peer-to-Peer (Component Interaction)

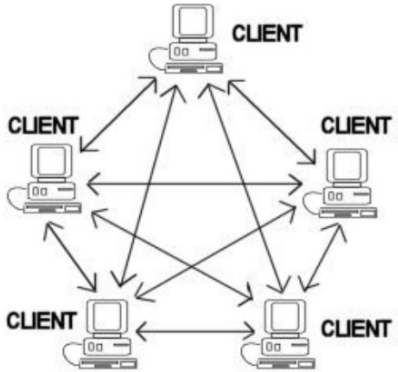
Problem: how to have a network in which there is no single point of failure?

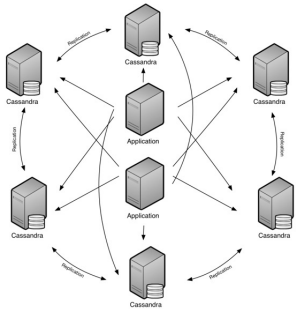
Completely decentralized

Although sometimes there still is a central server that helps peers find each other

Advantage:
- No single point of failure
- Extremely scalable

Disadvantages
- Clients contribute a lot of resources (prohibitive on mobile phones)
- Can't bad bad actors



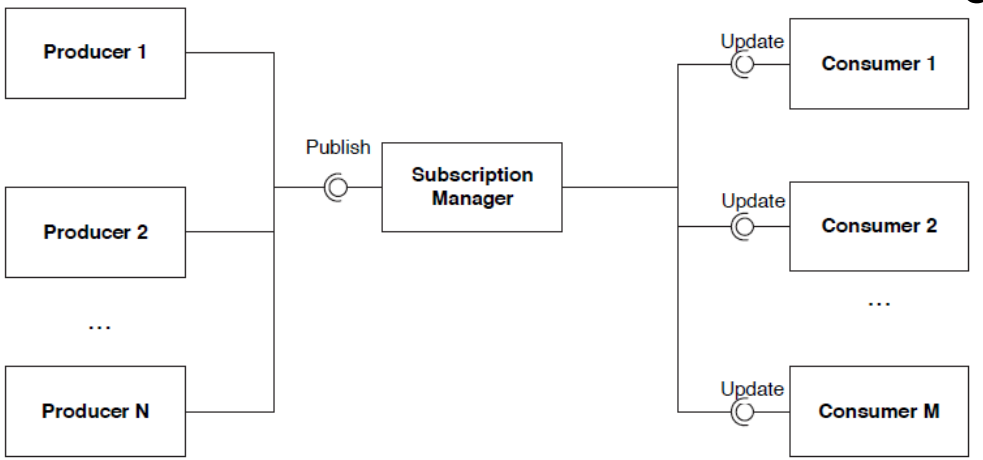


e.g. architecture for distributed databases

Publish/Subscribe (Component Interaction)

Problem: Event consumers and producers should be decoupled. Many consumers should receive events from one producer

Producers publish events



Subscription Manager routes

Consumers register interest in events from Producer via subscriptions

E.g., topic-based ("stocks/msft") or content-based ("notify if stocks.msft < 10$")

Canonical example?

- Stock quotes

?

?

Viewpoint: **Adaptation**

… the system is viewed as a core part that remains invariable and adaptable part that can change over time or in different versions.

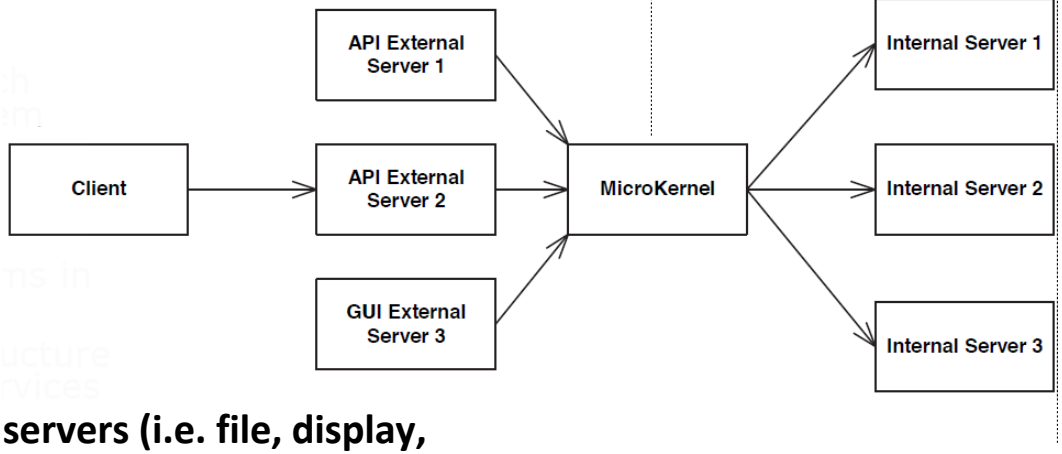
Concerns

- How can a system adapt to evolution over time or to multiple different versions of a basic architecture?
- What is the system functionality that is more likely to change and what will possibly remain invariable?
- How do the invariable parts communicate with the adaptable parts?
- How are the quality attributes of modifiability, reusability, evolvability, and integrability supported?

Pattern: **Microkernel**

Problem: system family in which different versions of a system need to be supported

Realizes services that all systems in the family need.
Plug-and-play infra for system-specific services



Only accessed by the Microkernel

External servers (i.e. file, display, network) are the only way for clients to access the microkernel

Canonical example? Microkernel OS

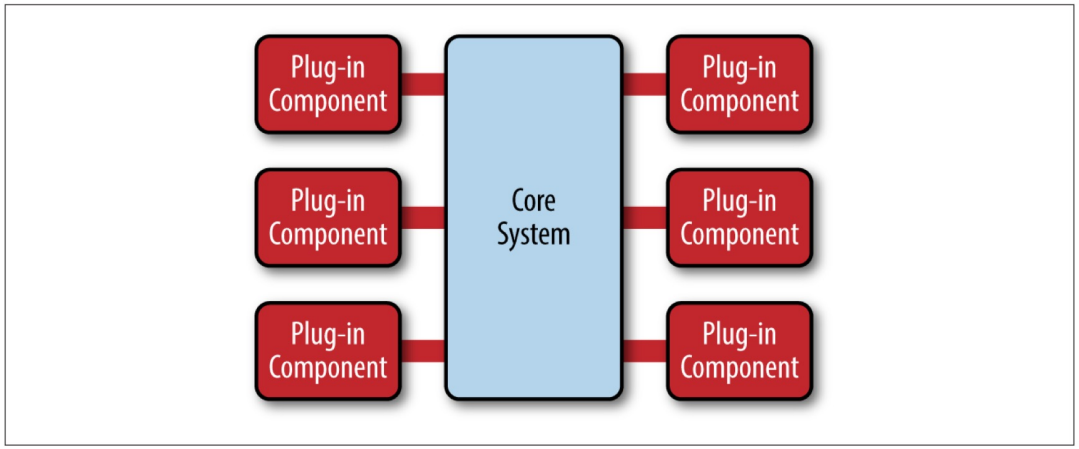
?

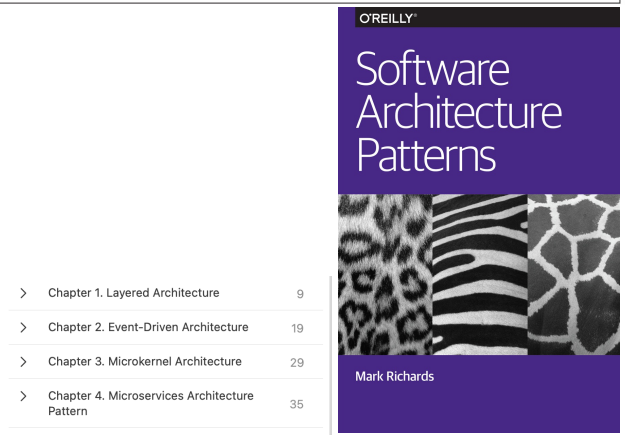
?

Pattern: **Plugin Architecture**

Plugins run in the same memory space as the original system

Classical examples:
- Eclipse IDE
- Web browsers





Viewpoint: **Data Flow**

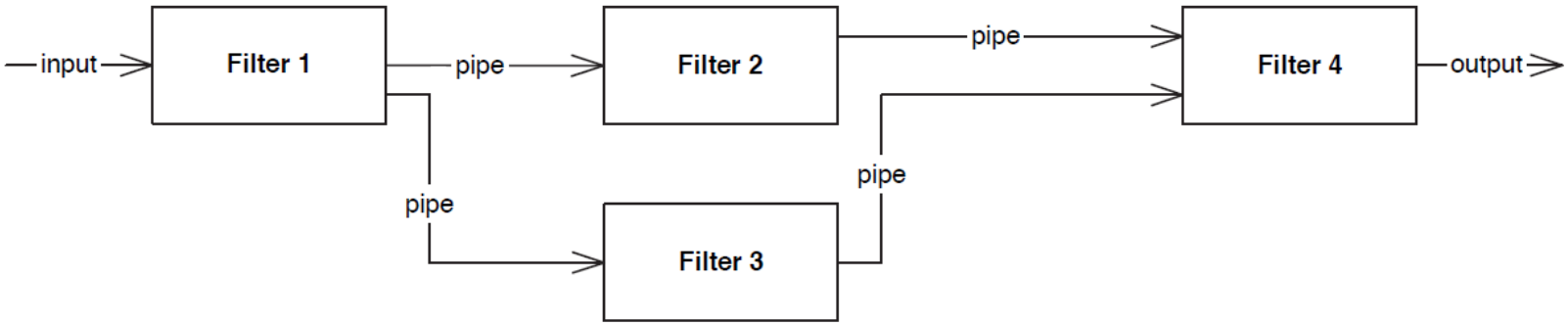
… system is seen as a number of subsequent transformations upon streams of input data

Concerns

- What are the elements that perform the transformations?
- What are the elements that carry the streams of data?
- How are the two aforementioned types of elements connected to each other?
- How are the quality attributes of performance, scalability, modifiability, reusability, and integrability supported?

Pipes and Filters (Data Flow)

Used if a complex task can divided into subtasks, each realized as an independent computation.



Filters
- transform input data
- consume and produce data incrementally
- are composed using pipes which stream data

Alternative: batch processing

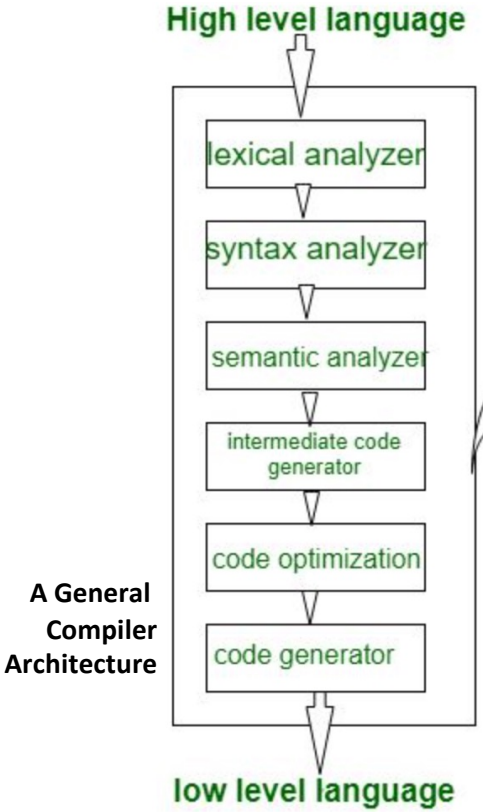
Canonical example?
- UNIX pipes and filters
- Classical compiler arch

?

?

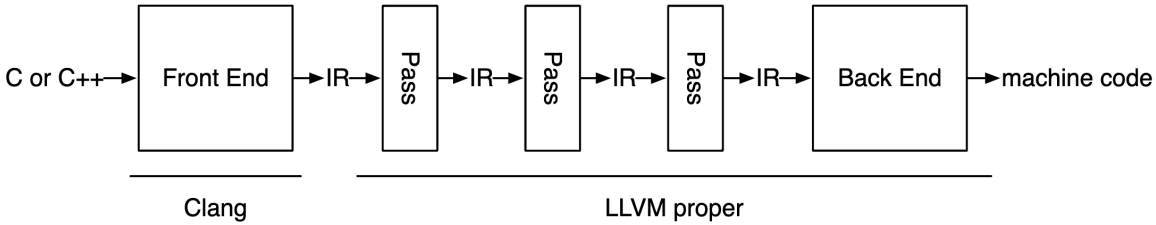
https://docs.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters

e.g. Architecture of the LLVM compiler infrastructure



A General Compiler Architecture

Architecture of LLVM
https://www.cs.cornell.edu/~asampson/blog/llvm.html



* Could be seen also as layers

Advantage of pipes and filters: versatility

Even though the UNIX system introduces a number of innovative programs and techniques, no single program or idea makes it work well. Instead, what makes it effective is the approach to programming, a philosophy of using the computer. Although that philosophy can't be written down in a single sentence, at its heart is the idea that the power of a system comes more from the relationships among programs than from the programs themselves. Many UNIX programs do quite trivial things in isolation, but, combined with other programs, become general and useful tools.

The Unix Programming Envirnonment

#System management

```
find /usr/bin |                 # produce
sed 's:.*/::' |                 # transform
grep -e '^w'  |                 # filter
xargs -I {} echo {}             # consume
```

#Data Analysis

```
cat grades | sort | uniq -c | sort -nr
```

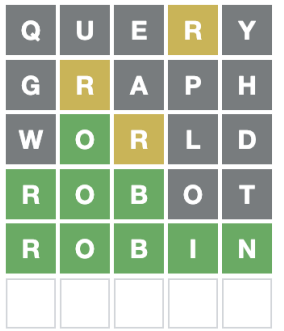
#NLP

```
cat /usr/share/dict/words | grep -e "^rob[^r].$" | grep -vE "q|u|e|y|g|a|p|h|w|l|d|t" | wc -l
```

Disadvantages?

Brian Kernigan on pipes in Unix

Advantage of pipes and filters: versatility




```
cat /usr/share/dict/words | wc -l
  235886

cat /usr/share/dict/words | grep -e "^...[^r].$" | grep r |  grep -vE "q|u|e|y" | wc -l
  885

cat /usr/share/dict/words | grep -e "^.[^r].[^r].$" | grep r | grep -vE "q|u|e|y|g|a|p|h" | wc -l
  81

cat /usr/share/dict/words | grep -e "^.o[^r][^r].$" | grep r | grep -vE "q|u|e|y|g|a|p|h|w|l|d" | w
  9

cat /usr/share/dict/words | grep -e "^rob[^r].$" | grep -vE “q|u|e|y|g|a|p|h|w|l|d|t" | wc -l

  1
```

Advantage of pipes and filters: versatility

Even though the UNIX system introduces a number of innovative programs and techniques, no single program or idea makes it work well. Instead, what makes it effective is the approach to programming, a philosophy of using the computer. Although that philosophy can't be written down in a single sentence, at its heart is the idea that the power of a system comes more from the relationships among programs than from the programs themselves. Many UNIX programs do quite trivial things in isolation, but, combined with other programs, become general and useful tools.

The Unix Programming Envirnonment

#System management

```
find /usr/bin |                 # produce
sed 's:.*/::' |                 # transform
grep -e '^w'  |                 # filter
xargs -I {} echo {}             # consume
```

#Data Analysis

```
cat grades | sort | uniq -c | sort -nr
```

#NLP

```
cat /usr/share/dict/words | grep -e "^rob[^r].$" | grep -vE "q|u|e|y|g|a|p|h|w|l|d|t" | wc -l
```

Disadvantages? Lack of Interactivity. Ease of making mistakes. Not for beginners.

Viewpoint: **Distribution**

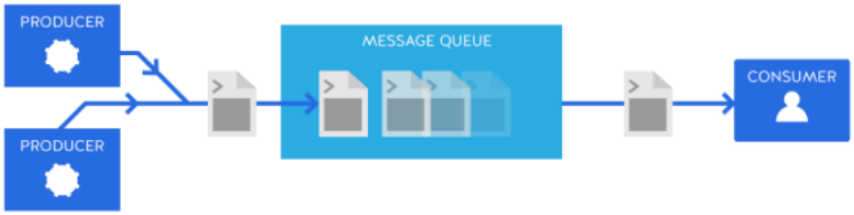
… tackles concerns about disseminating components in a networked environment.

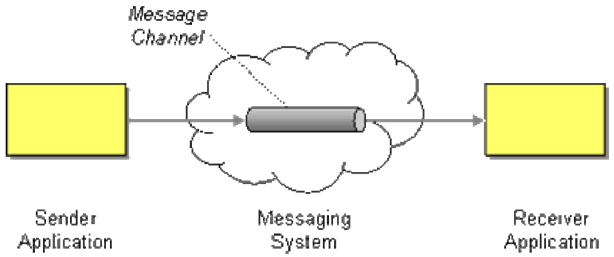
Concerns:

- How do the distributed components interact with each other?
- How are the distributed components decoupled from each other?
- How are the quality attributes of interoperability, location-transparency, performance, and modifiability supported?

Message queue

Problem: Invocations need to be sent over a network and be robust to outages of the receiver; a





Producer puts request into a queue - and can continue to operate

Consumer pulls requests from the queue and processes

Senders and receivers are decoupled in time and space

Canonical example?

- Bank transaction system

?

?

Event Broker

Problem: how do we update a consumer about a state change while keeping them decoupled from the producers ?

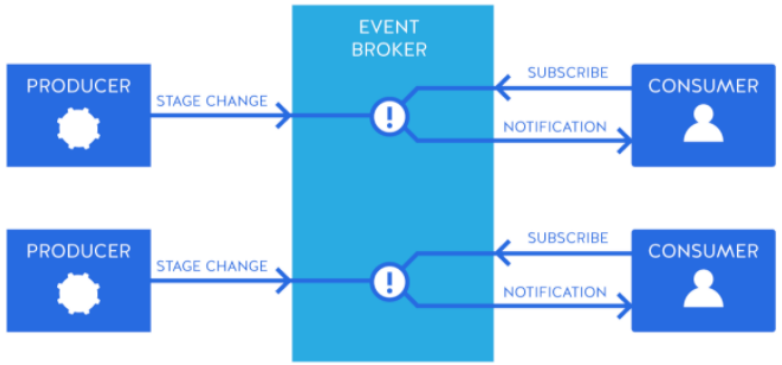
Event-Driven Architecture approach

Consumers “subscribe” to events and receive notifications when they occur.

Producer is not aware of the consumer who receives it

The event **broker** in the middle
- allows both parties to scale and evolve in a loosely coupled manner
- might stores events or not

Challenges: Broker should be
- scalable
- high performant
- fault-tolerant



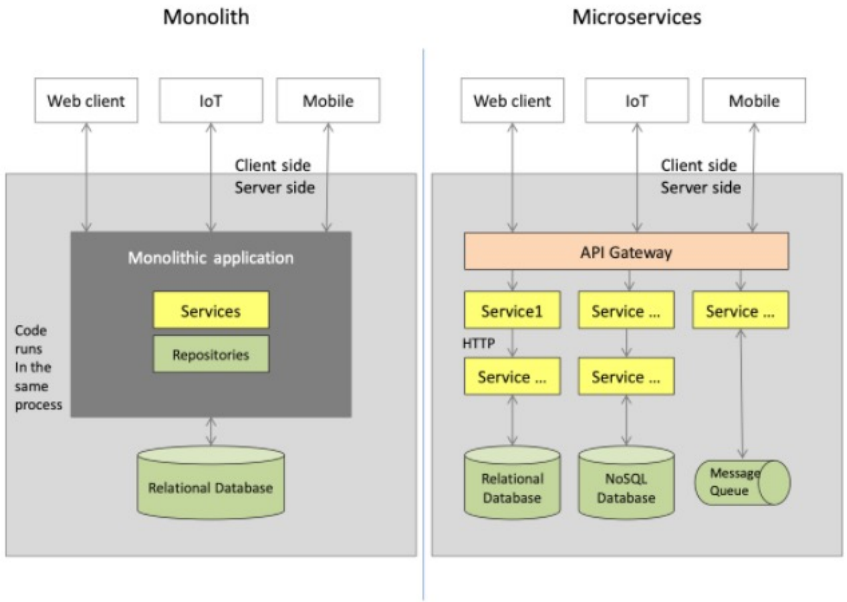
http://radar.oreilly.com/2015/02/variations-in-event-driven-architecture.html

Microservices

Problem: You need a highly available and scalable system / you have hundreds of developers and you need to give evolve differently

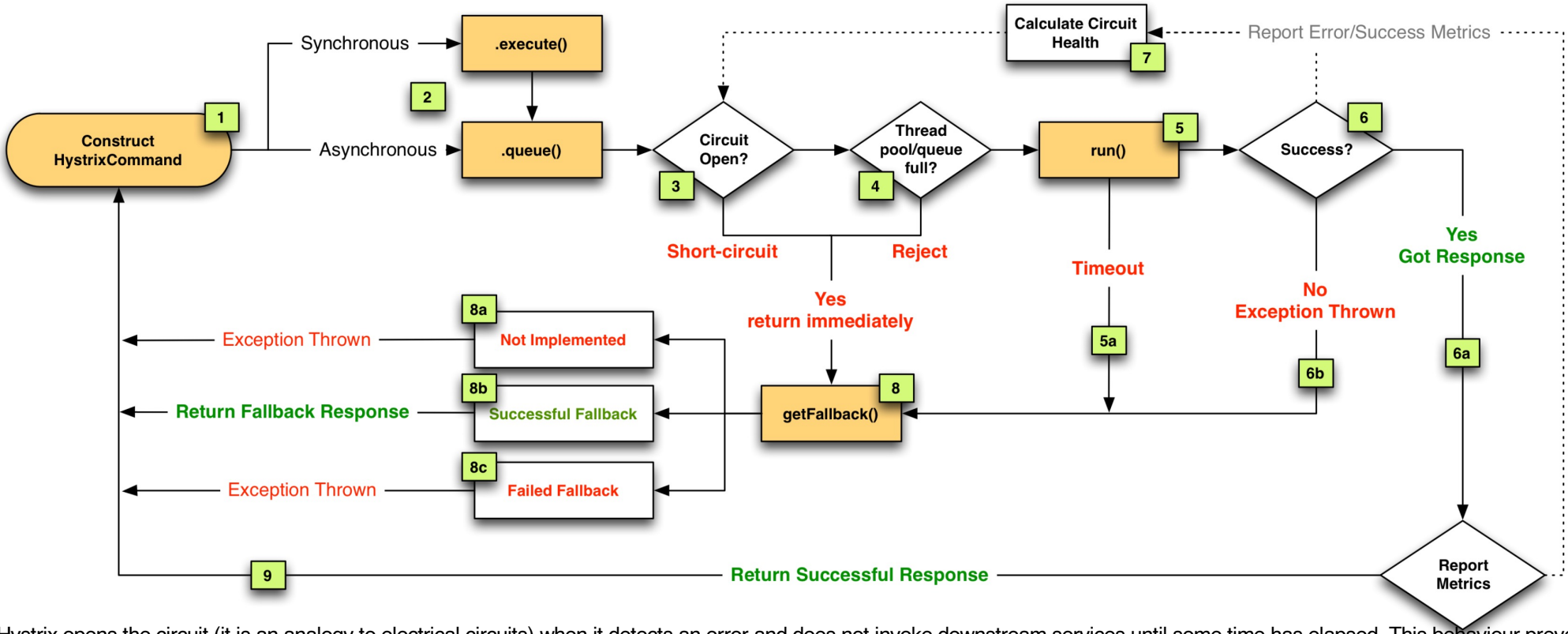
Services
- run independently
- deployed independently
- can use different technologies
- can be scaled independently

Challenges
- communication complexity (CAP theorem: consistency, availability, and partitioning… => eventual consistency)
- performance
- [return of the monolith]



https://developer.ibm.com/articles/challenges-and-benefits-of-the-microservice-architectural-style-part-1/

e.g. Netflix Error Handling with Hystrix



Hystrix opens the circuit (it is an analogy to electrical circuits) when it detects an error and does not invoke downstream services until some time has elapsed. This behaviour preve

Monolith

Martin Fowler - MonolithFirst

DHH - Majestic monolith - you are not Amazon or Google, embrace your monolith

Hybrid - microservices for the subsystems where consistency is not important.

Scalability via load balancing
https://martinfowler.com/articles/microservices.html

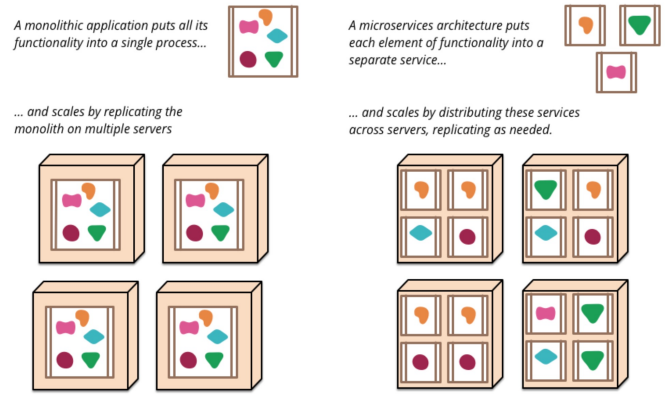


Figure 1: Monoliths and Microservices

Viewpoint: **Data-centered**

… system is seen as a persistent shared data stored that is accessed and modified by a number of elements.

Concerns

- How is the shared data store created, accessed, and updated?
- How is data distributed?
- Is the data store passive or active, i.e. does it notify its accessors or are the accessors responsible of finding data of interest to them?
- How does the data store communicate with the elements that access it?
- Do the accessor elements communicate indirectly through the shared data or also directly with each other?
- How are the quality attributes of scalability, modifiability, reusability, and integrability supported?

Shared Repository

Problem: complex data needs to be shared among components, including persistency.

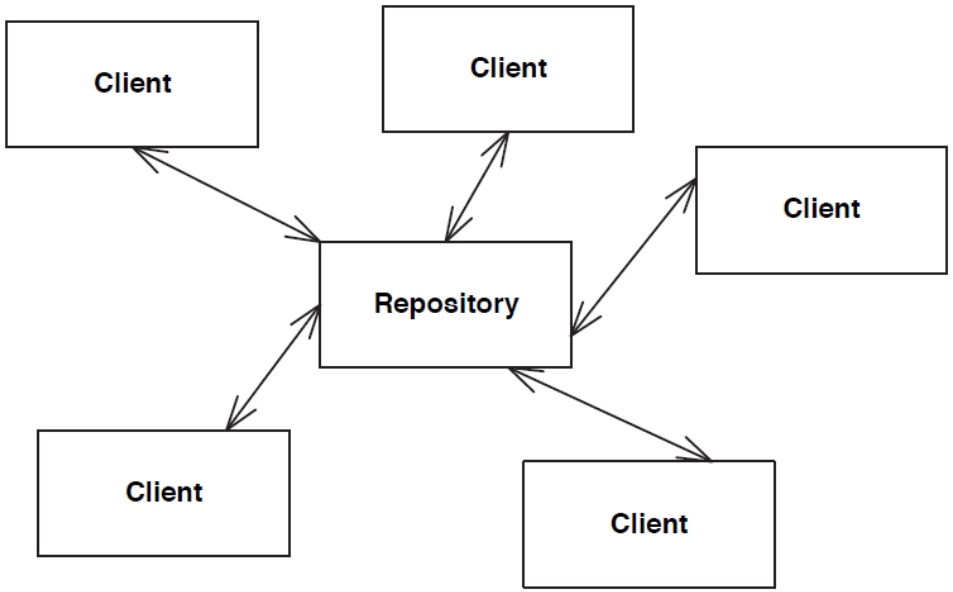
Central shared data repository accessed by all other independent components.

Shared repository offers means for accessing data

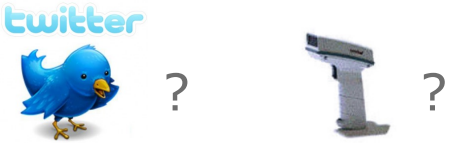
Advantage:
- you don't have to send the data around (compare with micro-services?)_

Challenges:
scalability, data consistency, resource contention handling, … as needed

Canonical example:
Database-centered system
IDE



https://www.hillside.net/plop/plop98/final_submissions/P24.pdf



Viewpoint: **Layers**

… the system is viewed as a complex heterogeneous entity decomposed in interacting parts.

Concerns:

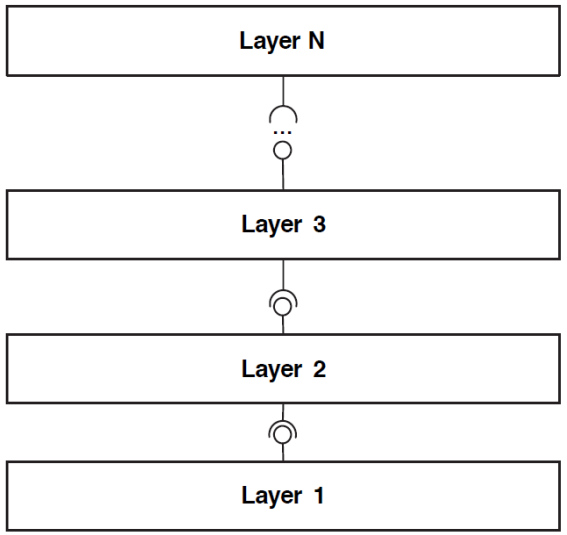
- What are the parts that make up the whole system?
- How do these parts interact with each other?
- How do the parts perform their functionality and still remain decoupled from each other?
- How are the quality attributes of modifiability, portability, and performance supported?

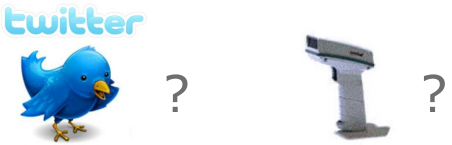
Layered Architecture

Used if system can be decomposed such that high-level components depend on low level

A layer
- **may only access** the layer(s) below it
  - Open = can depend on any lower layer
  - Closed = only on the next one
- **provides services** to the layer above it through a well defined interface
- contains elements work on the **same abstraction level**

Canonical examples:
- OSI stack (physical, data, transport, session, presentation, application)
- Android OS





Alternative? =>

Big Ball of spaghetti

Casually, even haphazardly, structured system
Dictated more by expediency than design

***De-facto standard* software architecture**
Seldom discussed :)

Reasons:
- cost
- experience
- visibility

a.k.a. spaghetti code

http://www.laputan.org/mud/

Viewpoint: **User Interaction**

… system is seen as a part that represents the user interface and a part that contains application logic

Concerns

- How is the user interface decoupled from the application logic?
- How are the quality attributes of usability, modifiability, and reusability supported?
- What is the data and the application logic that is associated to the user interface?

Model-View-Controller

Problem: multiple user interfaces are needed and these should be consistent

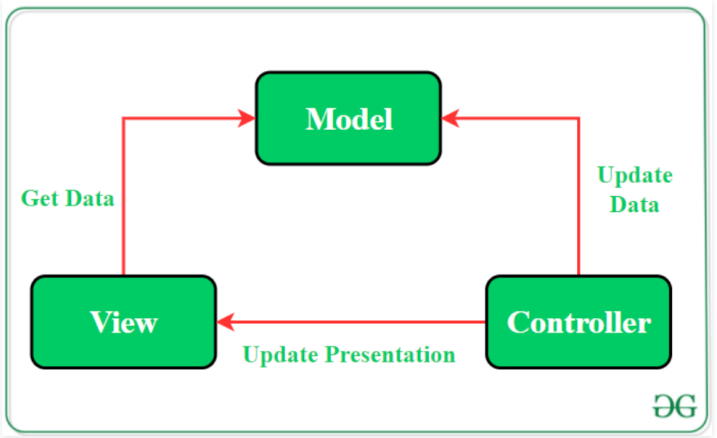
The Model encapsulates application data and logic. It does not depend on the other two.

A View displays (a part of) the data

A Controller is associated with a view and translates input into request on the model

Canonical example?

Smalltalk user interface architecture: a class for each of the three



https://martinfowler.com/eaaDev/uiArchs.html

?


?

There are many more patterns…

We only saw
- Subset of viewpoints
- Subset of patterns

Architectural patterns do not guide how to design for all external properties
- they support some main quality
- more qualities that need to be supported

E.g., if we need high scalability, it is not enough to apply the Client-Server pattern – we need **more detailed tools!**

Viewpoint: **Language Extension**

… concerned with how systems offer an abstraction layer to the computation infrastructure

Concerns

- How can a part of the system that is written in a nonnative language be integrated with the software system?

- How can the non-native part be translated into the native environment?

Interpreter

Problem: a language needs to be implemented inside an application.

An Interpreter provides a parsing and an execution environment

User-written scripts are interpreted at runtime

Canonical example?

- Build management system
- Game scripting engines (e.g. **Lua** based scripts)

?


?

To be continued next time

4. Architectural Tactics

Surgical means of supporting qualities

Bass, Clements, Kazman: **Software Architecture in Practice.** Addison-Wesley

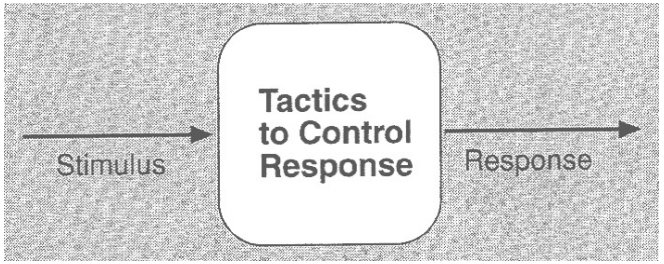
- 4th Edition (2021). Section 4.2 (availability), Section 8.2 (modifiability), Section 9.2 (performance), Section 11.2 (security)

- **3rd Edition** (2013). Sections 5.2, 7.2, 8.2, 9.2.

Architectural tactics

Architectural decisions that influence the achievement of quality attribute

They directly affect system's response to a stimulus



E.g., Heartbeat to control availability

Characteristics

- Capture what architects do in practice
- May influence more than one quality attribute
    - Since quality attributes are interdependent

Collection of tactics = strategy

Relationship to Patterns

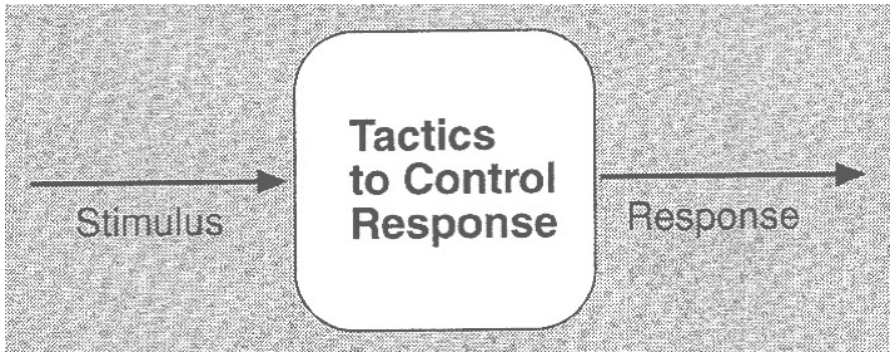
A pattern (or several) might not completely solve the architect's problem completely.

If no pattern exists for a given goal, tactics can construct a solution from "first principles"

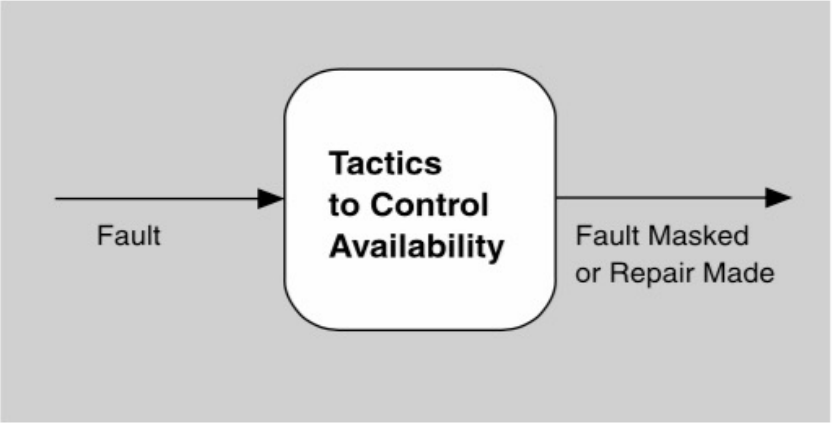
Categories of tactics

according to (main) quality attribute

1. **Availability**
2. **Modifiability**
3. **Performance**
4. **Security**
5. Testability
6. Usability
7. Interoperability



Availability tactics



A property of software that…

it is there and ready to carry out its task when it is required.

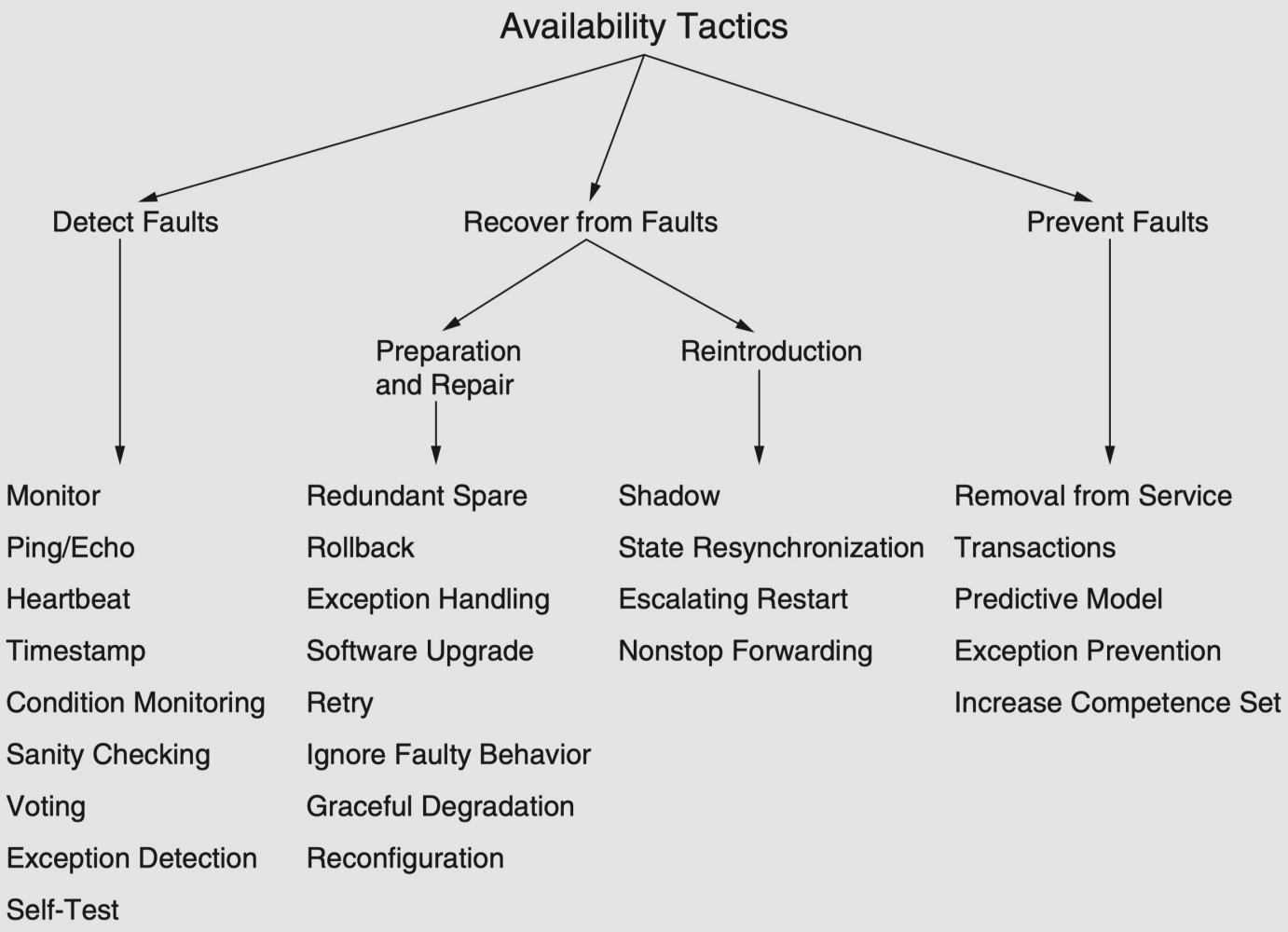
Encompasses what is normally called **reliability** but **adds to it the notion of recovery** (repair).

Availability tactics … keep *faults* from becoming *failures*!

A **failure:** when the system no longer delivers a service consistent with its specification – **observable by the users**.

A **fault** is a **malfunction** that has the potential to cause a failure.

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



e.g. Netflix Histrix (detect + degradation)

Detect Faults

Ping/echo: request/response message pair exchanged between nodes, used to determine reachability and the round-trip delay through the associated network path.

Heartbeat: a periodic message exchange between a system monitor and a process being monitored.

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.

Voting: to check that replicated components are producing the same results. Comes in various flavors: replication, functional redundancy, analytic redundancy.

Detect Faults

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.

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.

Recover from Faults

A **protection group** is a group of nodes where one or more nodes are "active," with the remainder serving as redundant spares.

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).

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

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”.

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.

Recover from Faults

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.

Escalating Restart: recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected.

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.

Summary for Availability Tactics

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.

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

Exercise

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

POS availability scenarios

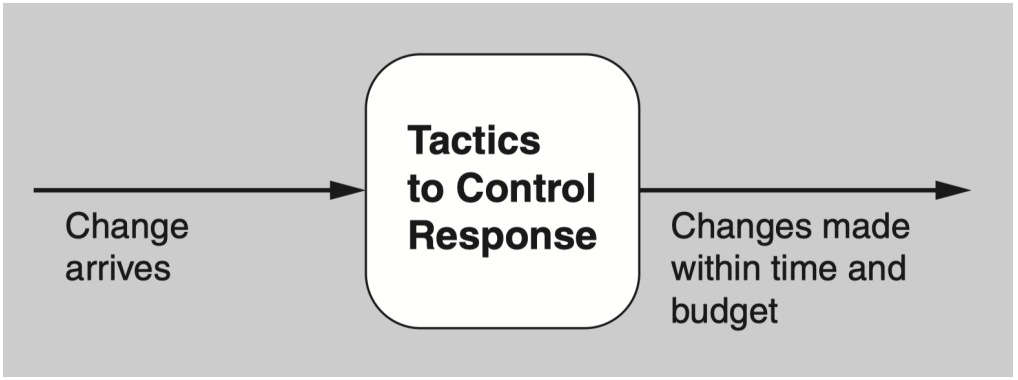
POS – Quality Attribute Scenario 2

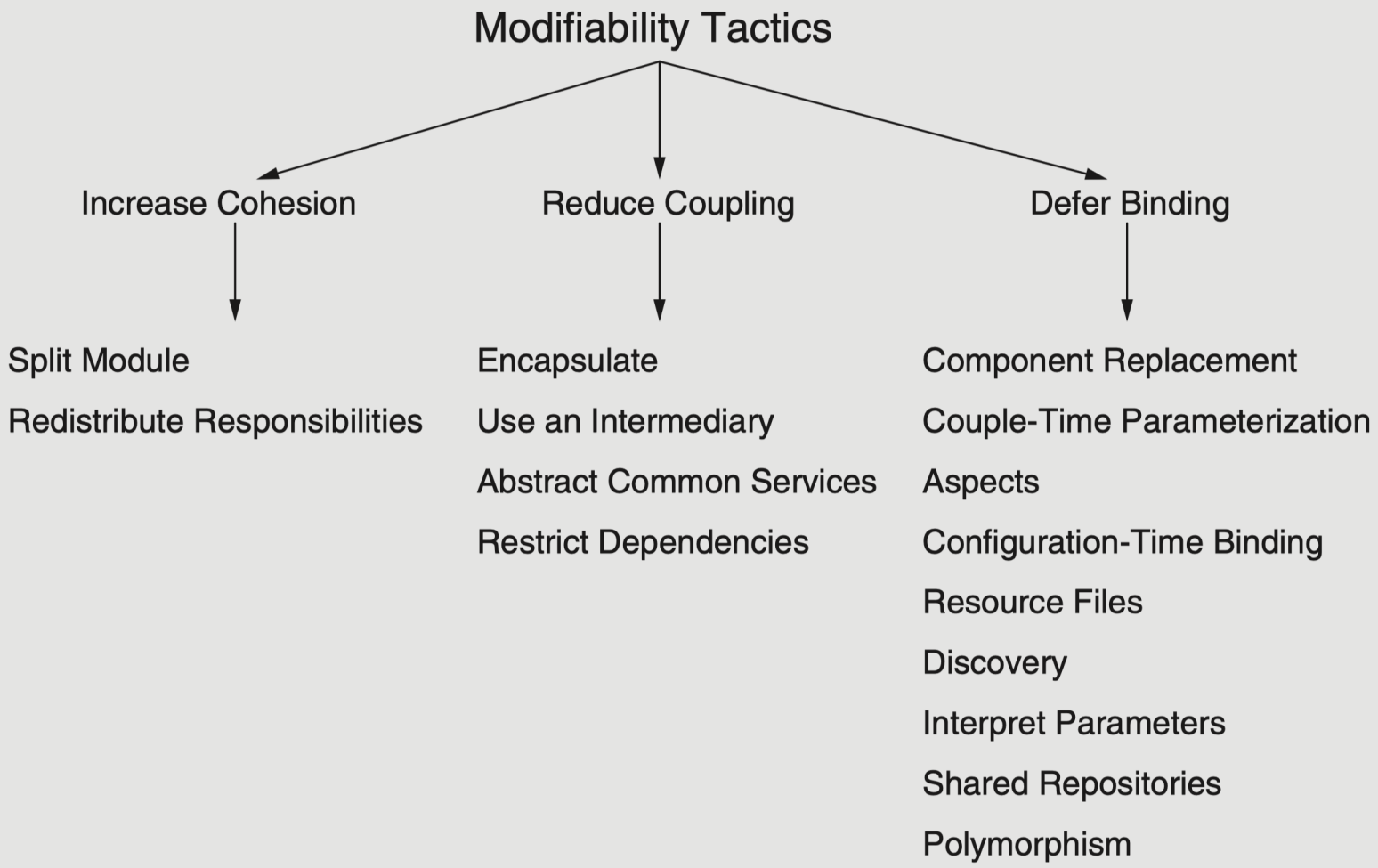
Scenario(s):		The inventory system fails and the failure is detected. The system continues to operate and queue inventory requests internally; issue requests when inventory system is running again
Relevant Quality Attributes:		Availability
Scenario Components	*Stimulus Source:*	Internal to system
	Stimulus:	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




Modifiability Tactics
Increase Cohesion
Reduce Coupling
Defer Binding
Split Module
Redistribute Responsibilities
Encapsulate
Use an Intermediary
Abstract Common Services
Restrict Dependencies
Component Replacement
Couple-Time Parameterization
Aspects
Configuration-Time Binding
Resource Files
Discovery
Interpret Parameters
Shared Repositories
Polymorphism

Increase Cohesion

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 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.

Restrict Dependencies: restricts the modules which a given module interacts with or depends on.

Defer Binding

In general, the later in the life cycle we can bind values, the better for modifiability

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

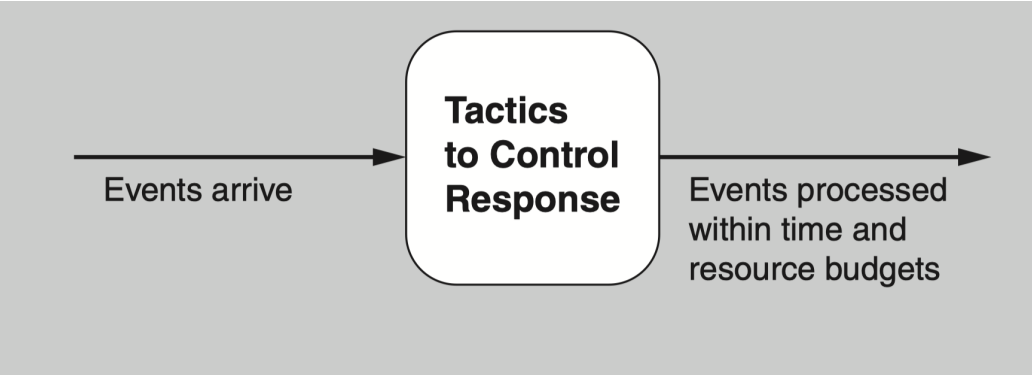
POS modifiability scenario

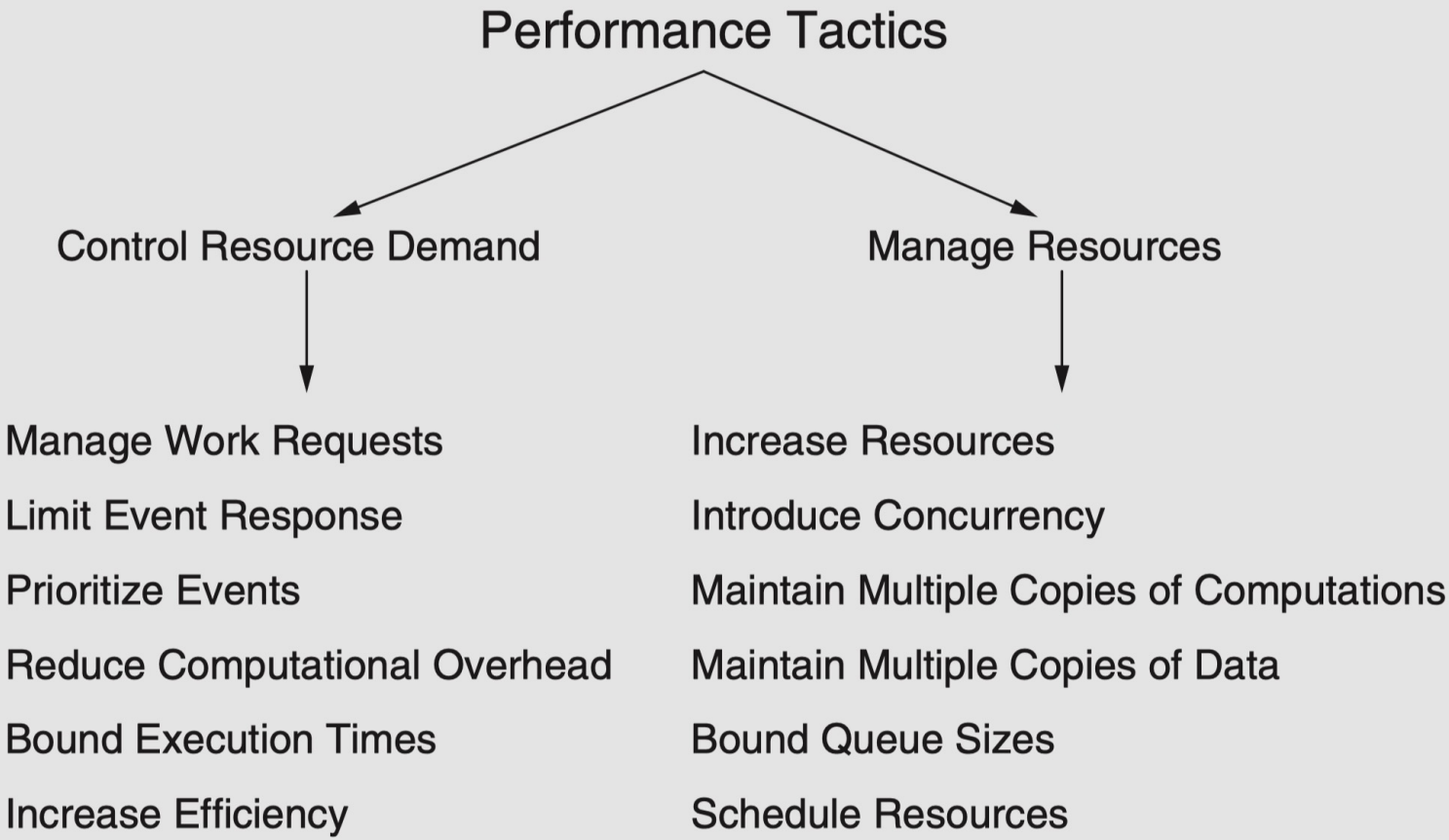
POS – Quality Attribute Scenario 3

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"

Think About: Which tactic would you use to handle the scenario?

Performance tactics




Performance Tactics
Control Resource Demand
Manage Resources
Manage Work Requests
Limit Event Response
Prioritize Events
Reduce Computational Overhead
Bound Execution Times
Increase Efficiency
Increase Resources
Introduce Concurrency
Maintain Multiple Copies of Computations
Maintain Multiple Copies of Data
Bound Queue Sizes
Schedule Resources

Control Resource Demands

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 Demands

Reduce Overhead: The use of intermediaries (important for modifiability) increases the resources consumed in processing an event stream; removing them improves latency. (e.g. Eclipse)

Bound Execution Times: Place a limit on how much execution time is used to respond to an event. (e.g. finding bugs)

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. (E.g. in-memory caching)

Schedule Resources: When there is contention for a resource, the resource must be scheduled.

- FIFO
- Fixed-priority scheduling
- Dynamic priority scheduling
- Static scheduling

POS performance scenario

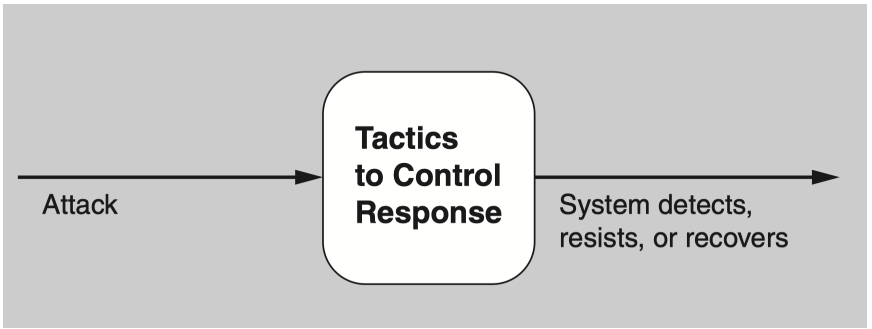
POS – Quality Attribute Scenario 4

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

Think About: Which tactic would you use to handle the scenario?

Security tactics

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)

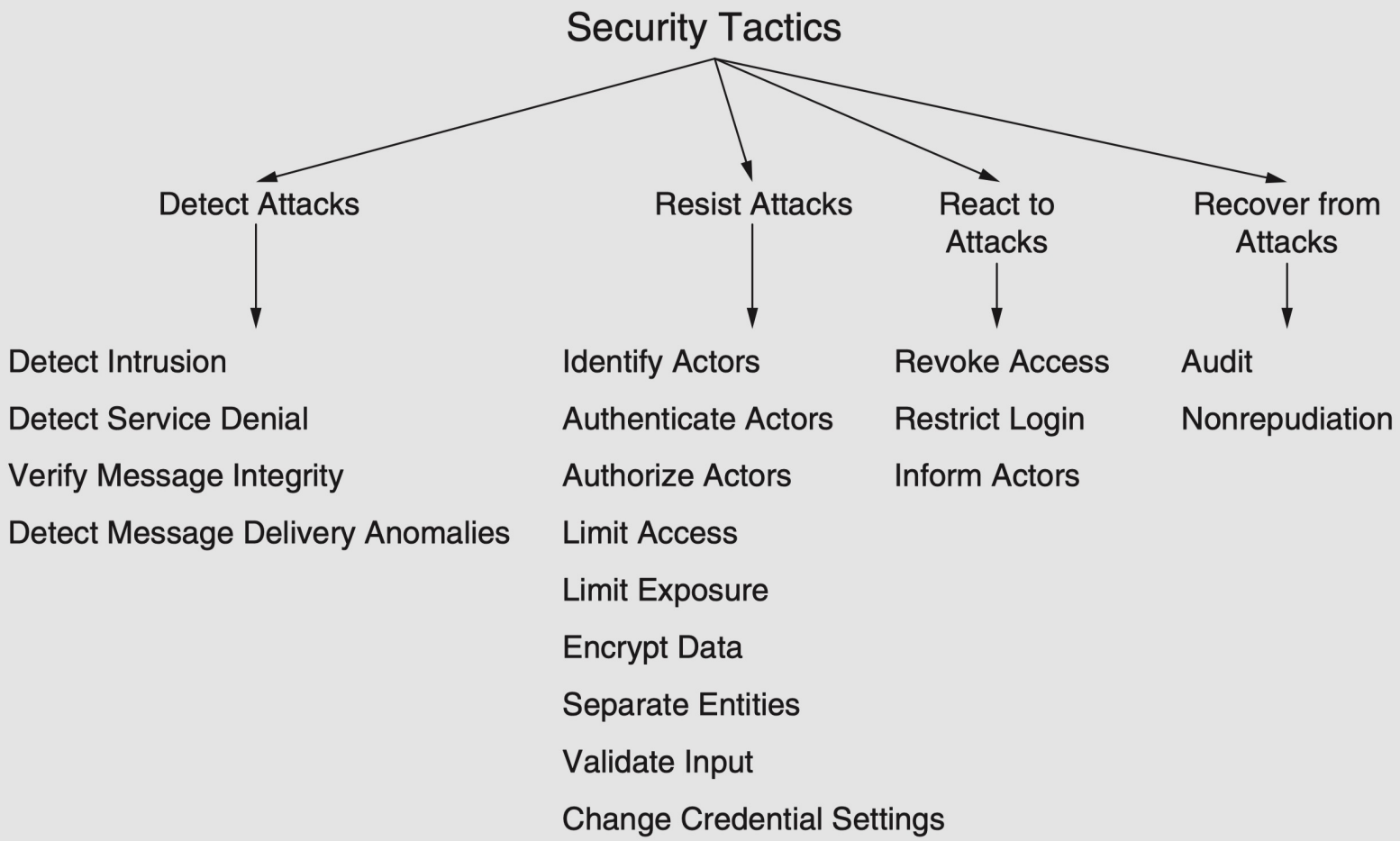


An action taken against a computer system with the intention of doing harm is called an **attack** and can take many forms.

Main Characteristics of Security

Security has three main characteristics:

- **Confidentiality** is the property that data or services are protected from unauthorized access. (e.g. 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. (e.g. your grade has not been changed since your instructor assigned it)
- **Availability** is the property that the system will be available for legitimate use. (e.g. a denial-of-service attack won't prevent you from ordering a book from an online bookstore)


Security Tactics
Detect Attacks
Resist Attacks
React to Attacks
Recover from Attacks
Detect Intrusion
Detect Service Denial
Verify Message Integrity
Detect Message Delivery Anomalies
Identify Actors
Authenticate Actors
Authorize Actors
Limit Access
Limit Exposure
Encrypt Data
Separate Entities
Validate Input
Change Credential Settings
Revoke Access
Restrict Login
Inform Actors
Audit
Nonrepudiation

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.

Resist Attacks

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 (e.g. not every user should be able to delete tables from DB)

Encrypt Data: apply some form of encryption to data and to communication.

React to Attacks

Revoke Access: limit access to sensitive resources, even for normally legitimate users and uses, if an attack is suspected.

Inform Actors: notify operators, other personnel, or cooperating systems when an attack is suspected or detected.

Recover from Attacks

Audit: keep a record of user and system actions and their effects, to help trace the actions of, and to identify, an attacker.

Summary

A “fake“ synthesis process

1) Choose **architectural style** according to quality requirements
2) Create overall, tentative structure described using the *3+1 approach*
3) Refine initial structure through **architectural tactics**
4) Consider architectural (and business) qualities
5) Maintain the **architectural backlog**