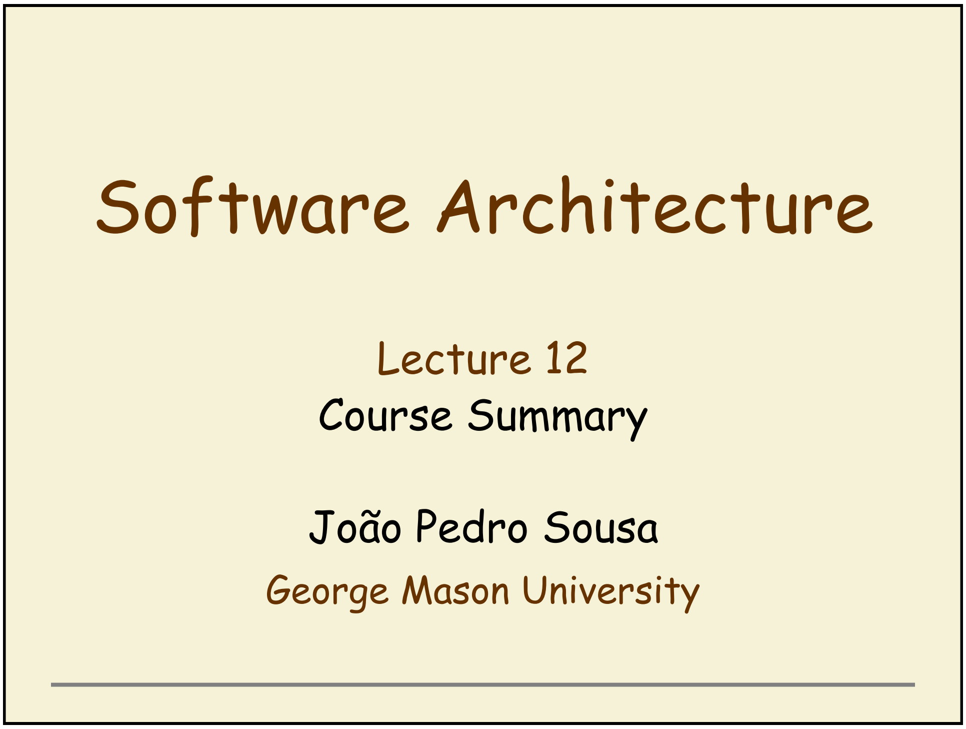


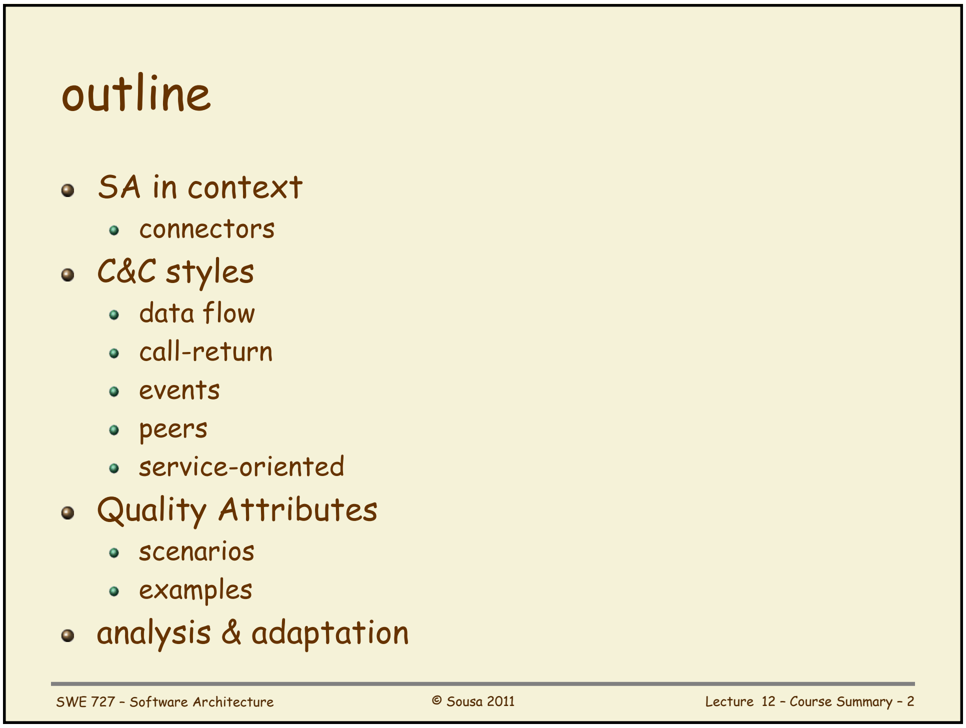
Software Architecture

Lecture 12

Course Summary

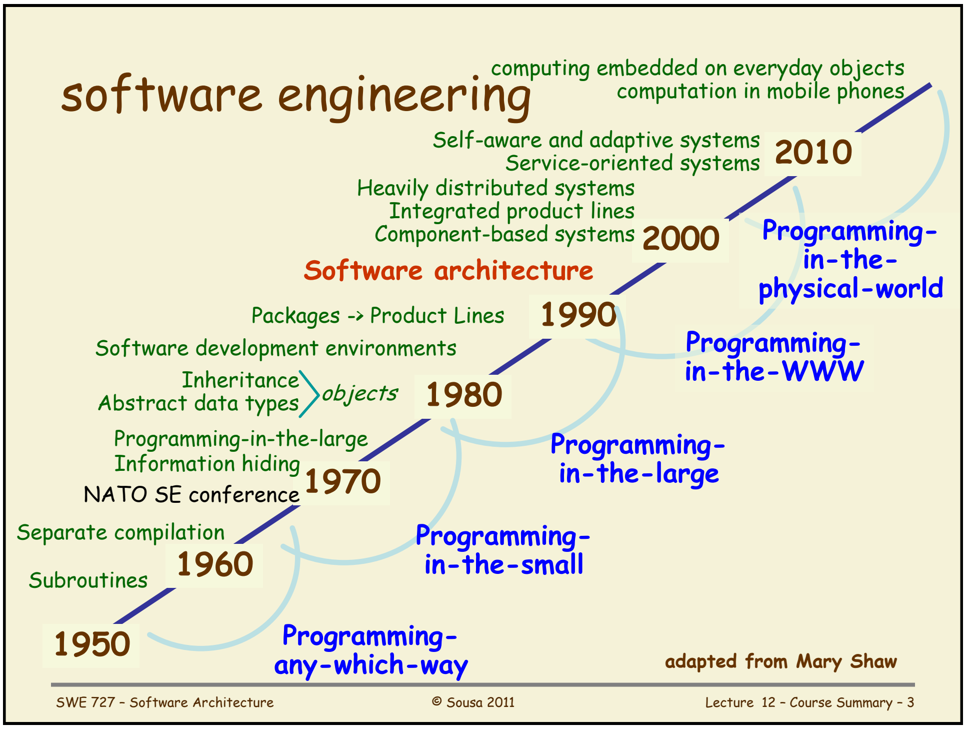
João Pedro Sousa

George Mason University



outline

- SA in context
  - connectors
- C&C styles
  - data flow
  - call-return
  - events
  - peers
  - service-oriented
- Quality Attributes
  - scenarios
  - examples
- analysis & adaptation



software engineering

computing embedded on everyday objects
computation in mobile phones

2010

Self-aware and adaptive systems
Service-oriented systems

Heavily distributed systems
Integrated product lines
Component-based systems

2000

Software architecture

Packages -> Product Lines

1990

Software development environments

Inheritance
Abstract data types
objects

1980

Programming-in-the-large
Information hiding
NATO SE conference

1970

Separate compilation

1960

Subroutines

1950

Programming-any-which-way

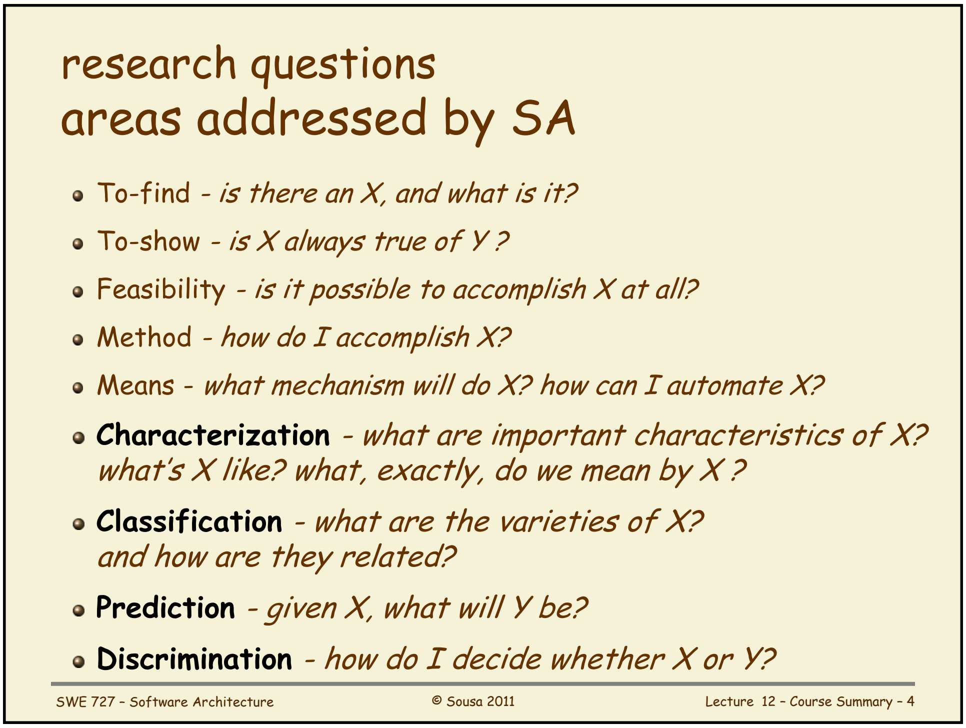
Programming-in-the-small

Programming-in-the-large

Programming-in-the-WWW

Programming-in-the-physical-world

adapted from Mary Shaw



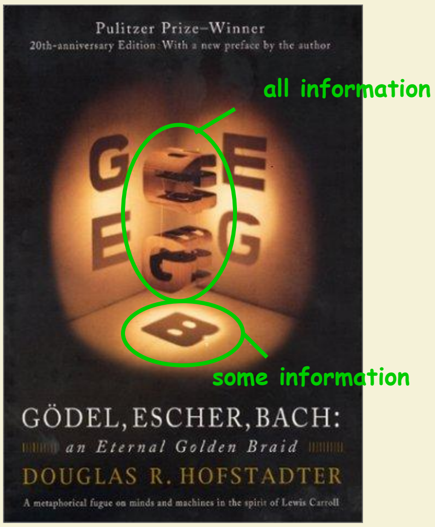
research questions
areas addressed by SA

- To-find - *is there an X, and what is it?*
- To-show - *is X always true of Y ?*
- Feasibility - *is it possible to accomplish X at all?*
- Method - *how do I accomplish X?*
- Means - *what mechanism will do X? how can I automate X?*
- **Characterization** - *what are important characteristics of X? what's X like? what, exactly, do we mean by X ?*
- **Classification** - *what are the varieties of X? and how are they related?*
- **Prediction** - *given X, what will Y be?*
- **Discrimination** - *how do I decide whether X or Y?*

one system, many views

- a view is a representation of a set of system elements and the relations among them
- not *all* system elements
- a view selects *element types* and *relation types* of interest, and shows only those

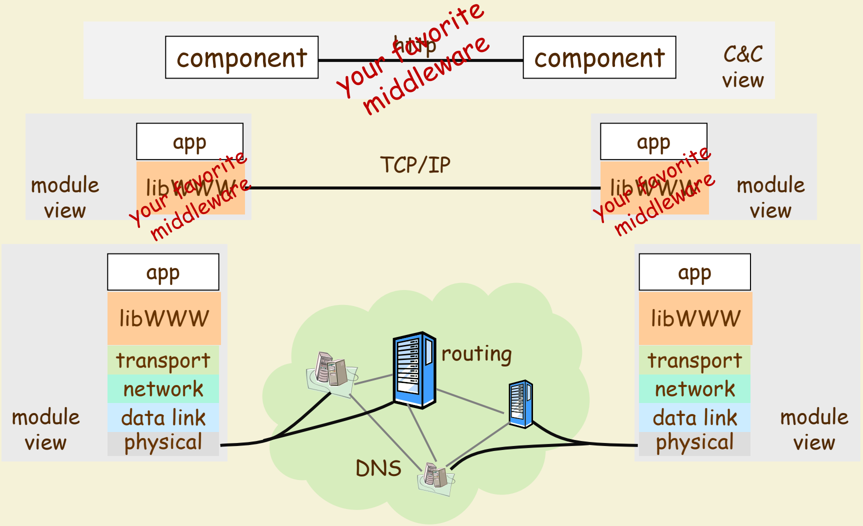
why?


Pulitzer Prize-Winner
20th-anniversary Edition With a new preface by the author
all information
some information
GÖDEL, ESCHER, BACH:
an Eternal Golden Braid
DOUGLAS R. HOFSTADTER
A metaphorical fugue on minds and machines in the spirit of Lewis Carroll

views help manage the complexity of describing an architecture

- viewtypes
  determine the kinds of things a view talks about
  - three primary viewtypes: module, C&C, allocation
- each viewtype has many styles
  - module: decomposition, generalization, layered, …
  - C&C: pipe & filter, client-server, pub-sub…
  - allocation: deployment, work assignment…

C&C
contributes notion of connector



C&C
many styles occur in practice

data flow
- batch sequential
- dataflow network (pipe & filter)
  - acyclic, fan-out, pipeline, Unix
- closed loop control

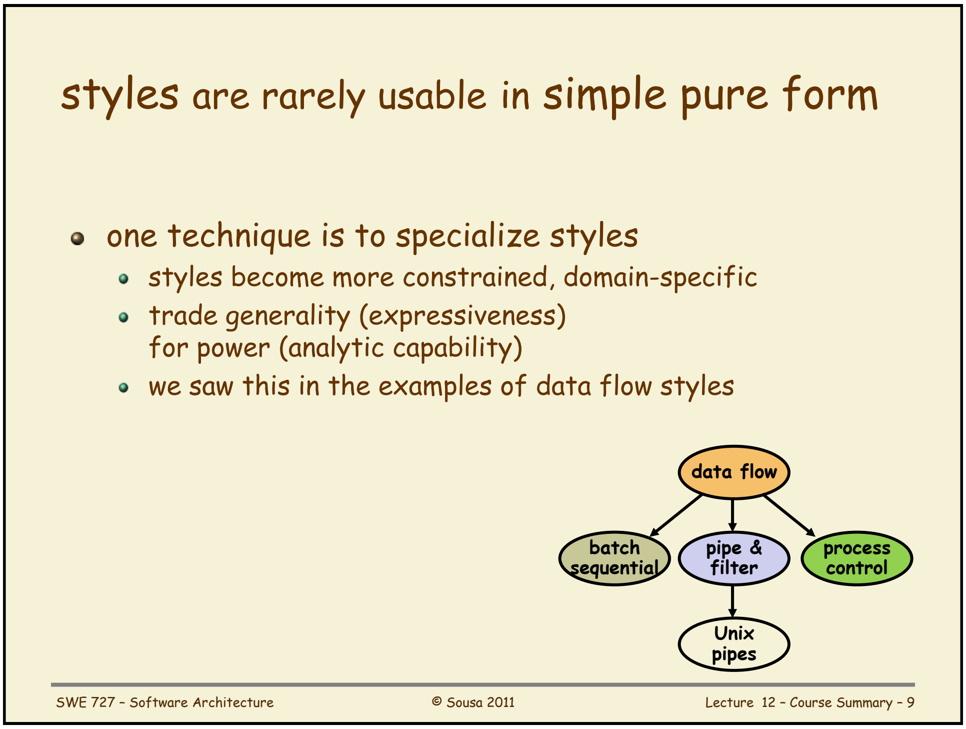
call-return
- main program/subroutines
- information hiding - objects
- stateless client-server
- SOA

interacting processes
- communicating peers
- event systems
  - implicit invocation
  - publish-subscribe

data-oriented repository
- transactional databases
  - stateful client-server
- blackboard
- modern compiler

data-sharing
- compound documents
- hypertext
- Fortran COMMON
- LW processes

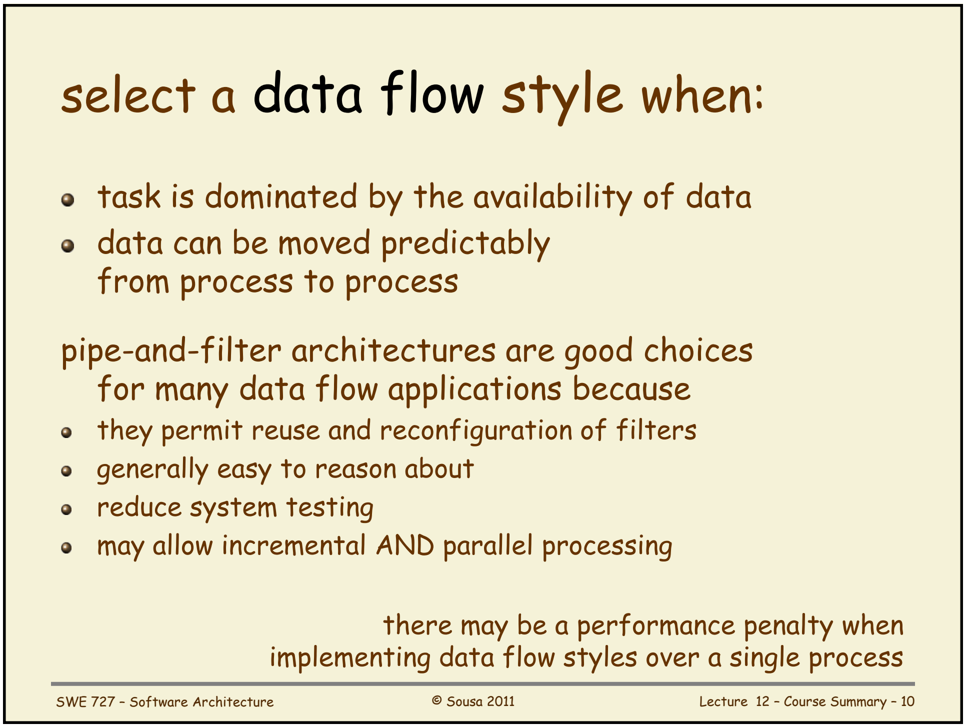
hierarchical
- tiers
  - interpreter
  - N-tiered client-server



styles are rarely usable in simple pure form

- one technique is to specialize styles
  - styles become more constrained, domain-specific
  - trade generality (expressiveness)
    for power (analytic capability)
  - we saw this in the examples of data flow styles

data flow
batch sequential
pipe & filter
process control
Unix pipes



select a data flow style when:

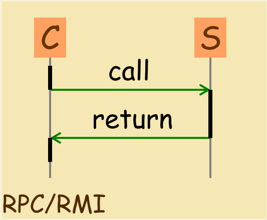
- task is dominated by the availability of data
- data can be moved predictably
  from process to process

pipe-and-filter architectures are good choices
for many data flow applications because

- they permit reuse and reconfiguration of filters
- generally easy to reason about
- reduce system testing
- may allow incremental AND parallel processing

there may be a performance penalty when
implementing data flow styles over a single process

select a call-return style when:



- task is dominated by single thread of control
- caller knows and cares about the identity of server
- low volume of data is transferred

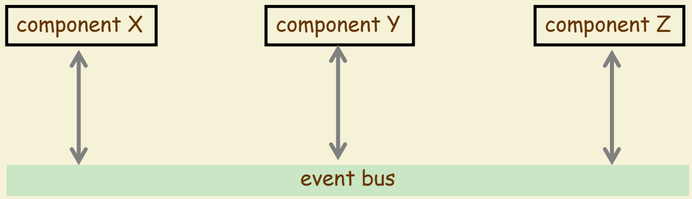
in distributed systems:

- it is fine to block the caller waiting for a reply
- the server is ready to process each request
- components and network are mostly reliable

interacting processes family tree

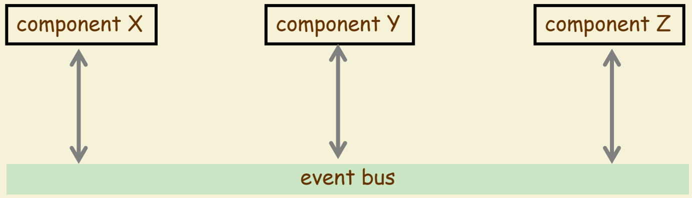
- communicating peers
  - asynchronous messages aka explicit events
    explicit wrt identifying the recipient
- event systems aka implicit events
  - events delivered to all interested components in some order
  - publish aka broadcast
  - publish-subscribe
    - interested components subscribe to events
    - interested components receive asynchronous message
  - implicit invocation
    - interested components register a callback method
    - upon the event, the method is invoked (call-return)

publish-subscribe & implicit invocation rely on event infrastructure



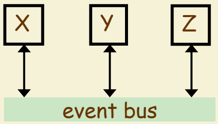
- identity of event recipients is unknown to senders
- order of event delivery is unknown
  - different event buses make different guaranties or no guaranties about ordering

many strategies for the event bus connector



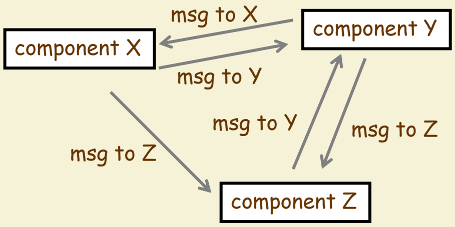
- push / pull
- component / bus event filtering
- call-return / asynchronous messages
- local / remote comms

event systems
easy to modify, hard to test



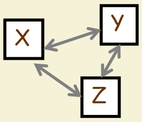
- QAs promoted
  due to decoupling and encapsulation
  - reuse
  - modifiability
  - scalability
- QAs inhibited
  - performance: hard to guarantee response time
  - testability: hard to test and reason about correctness
  - availability: possible to miss events (no recipients)

communicating peers
middle ground between call-return & events



	call-return	peers	events
identity of receiver is known	yes	yes	no
can prescribe/predict order	yes	yes	no
communication	synchronous	asynch	asynch
restrictions on topology	hierarchical	none	none

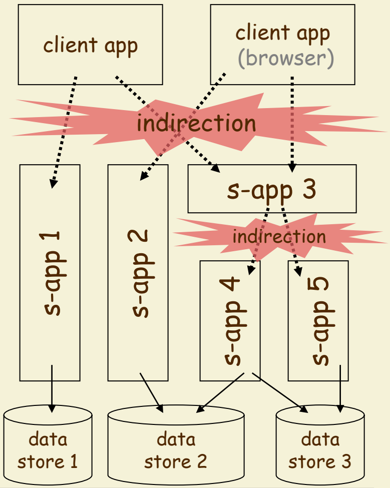
peer systems
responsive & robust but costly

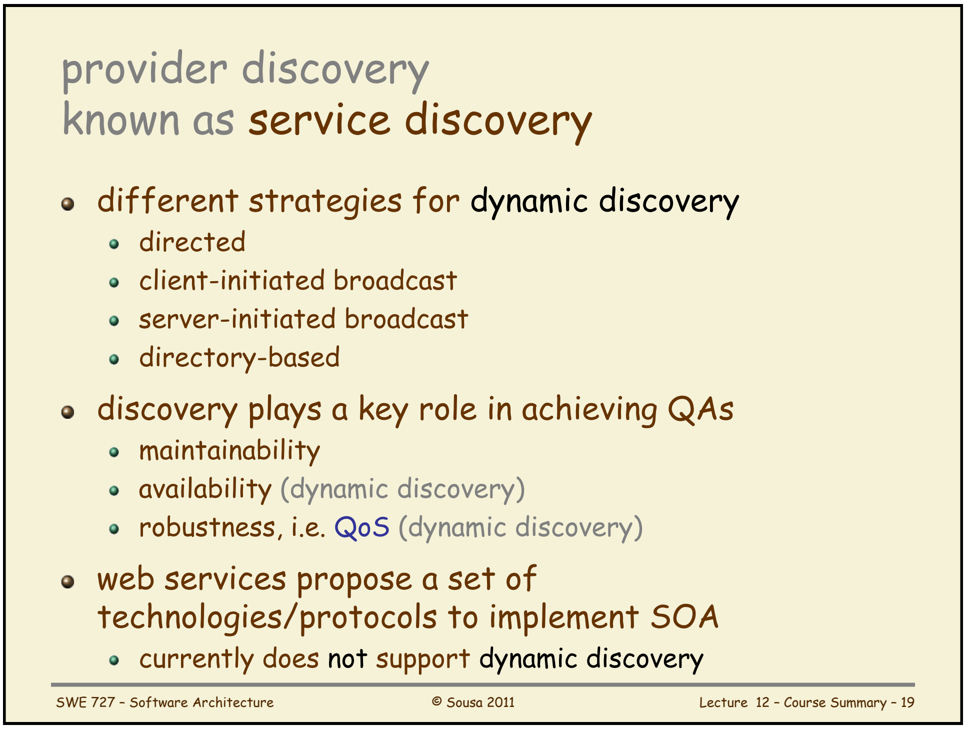


- QAs promoted
  - conceptual integrity
  - responsiveness
  - robustness
  - scalability
- QAs inhibited
  - development costs

Service Oriented Architectures
are evolution of tiered style

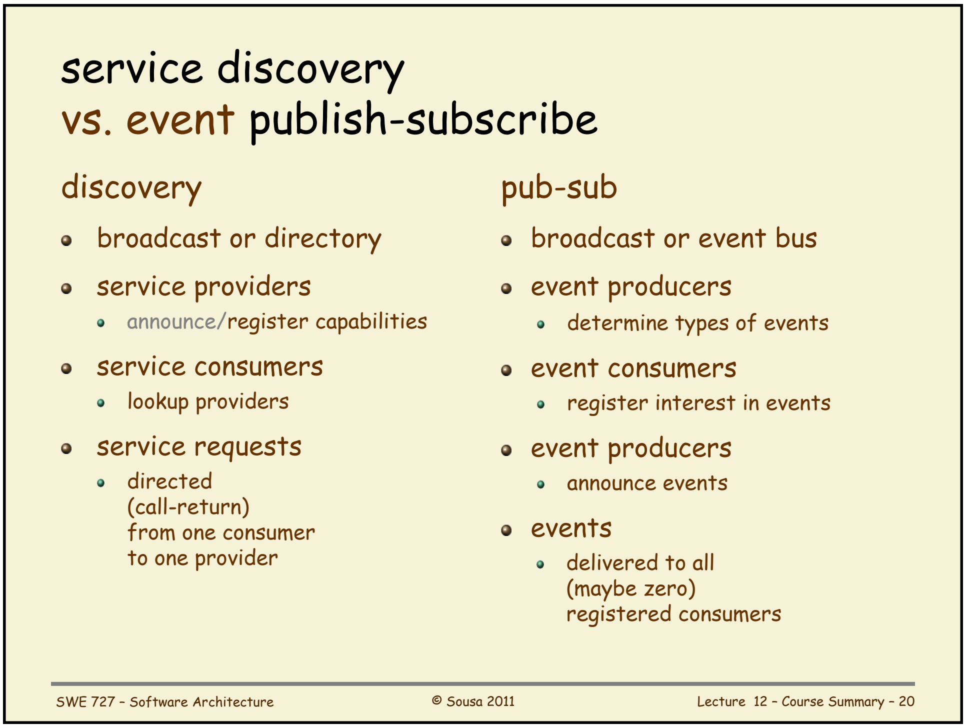
- complex apps already existed
  - normally all components hosted/maintained by the same organization
- SOA adds level of indirection
- service
  - is a unit of work
  - several candidate providers
    - maybe hosted by diff organizations
  - a provider may be *discovered*
    - before deployment, or
    - dynamically at run time





provider discovery known as service discovery

- different strategies for dynamic discovery
  - directed
  - client-initiated broadcast
  - server-initiated broadcast
  - directory-based
- discovery plays a key role in achieving QAs
  - maintainability
  - availability (dynamic discovery)
  - robustness, i.e. QoS (dynamic discovery)
- web services propose a set of technologies/protocols to implement SOA
  - currently does not support dynamic discovery



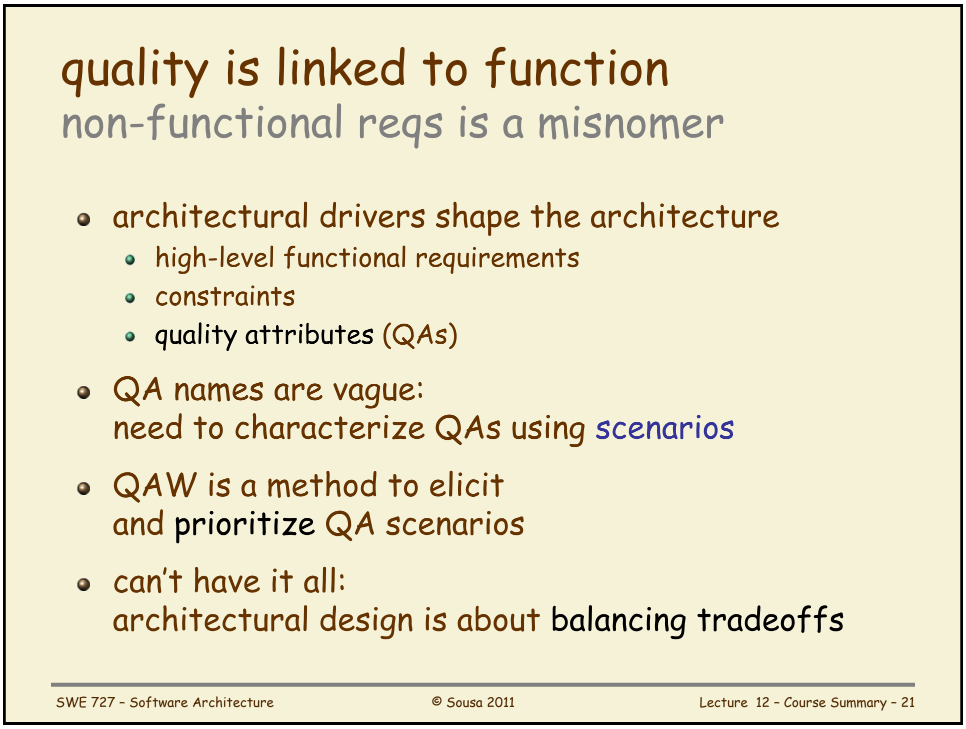
service discovery vs. event publish-subscribe

discovery

- broadcast or directory
- service providers
  - announce/register capabilities
- service consumers
  - lookup providers
- service requests
  - directed (call-return) from one consumer to one provider

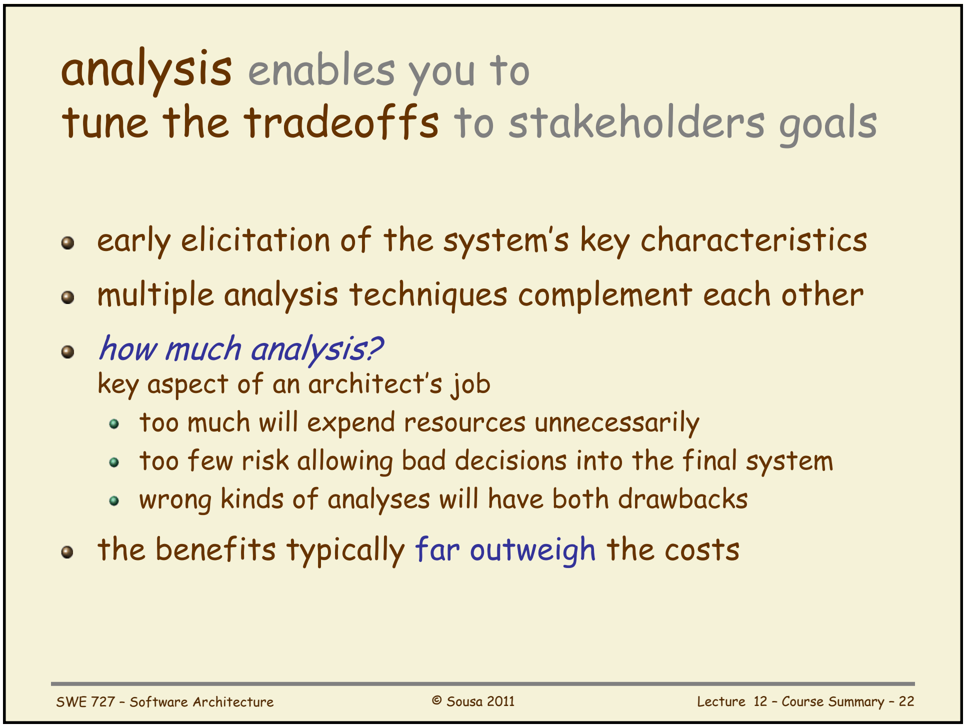
pub-sub

- broadcast or event bus
- event producers
  - determine types of events
- event consumers
  - register interest in events
- event producers
  - announce events
- events
  - delivered to all (maybe zero) registered consumers



quality is linked to function
non-functional reqs is a misnomer

- architectural drivers shape the architecture
  - high-level functional requirements
  - constraints
  - quality attributes (QAs)
- QA names are vague:
  need to characterize QAs using scenarios
- QAW is a method to elicit
  and prioritize QA scenarios
- can't have it all:
  architectural design is about balancing tradeoffs



analysis enables you to tune the tradeoffs to stakeholders goals

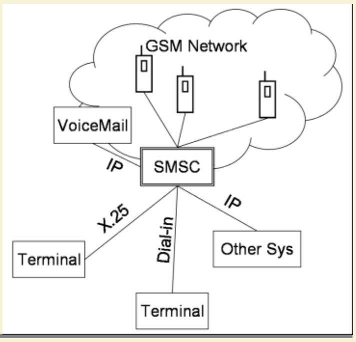
- early elicitation of the system's key characteristics
- multiple analysis techniques complement each other
- *how much analysis?*
  key aspect of an architect's job
  - too much will expend resources unnecessarily
  - too few risk allowing bad decisions into the final system
  - wrong kinds of analyses will have both drawbacks
- the benefits typically far outweigh the costs

automated analysis
key part of self-adaptation

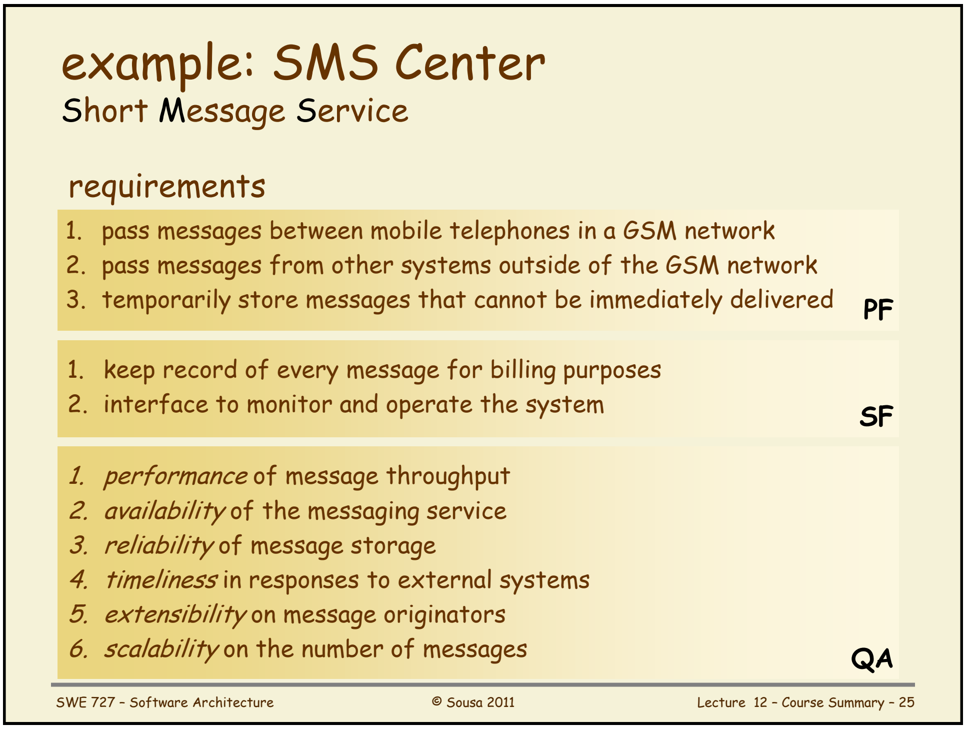
- traditional: manual changes
- emerging: automated changes
  aka self-adaptation
  - feedback control loop
    - automated discovery
    - automated QA analyses

example: SMS Center
Short Message Service

- system built by LogicaCMG (Netherlands)
  in the early 90's
- when the SMS market boomed in late 90's
  LogicaCMG dominated the market of SMS backend
  (mobile operators subcontracted them)
- why?
  architectural decisions
  based on QA analysis



Poort et al. WICSA 2005



example: SMS Center

Short Message Service

requirements

1. pass messages between mobile telephones in a GSM network
2. pass messages from other systems outside of the GSM network
3. temporarily store messages that cannot be immediately delivered

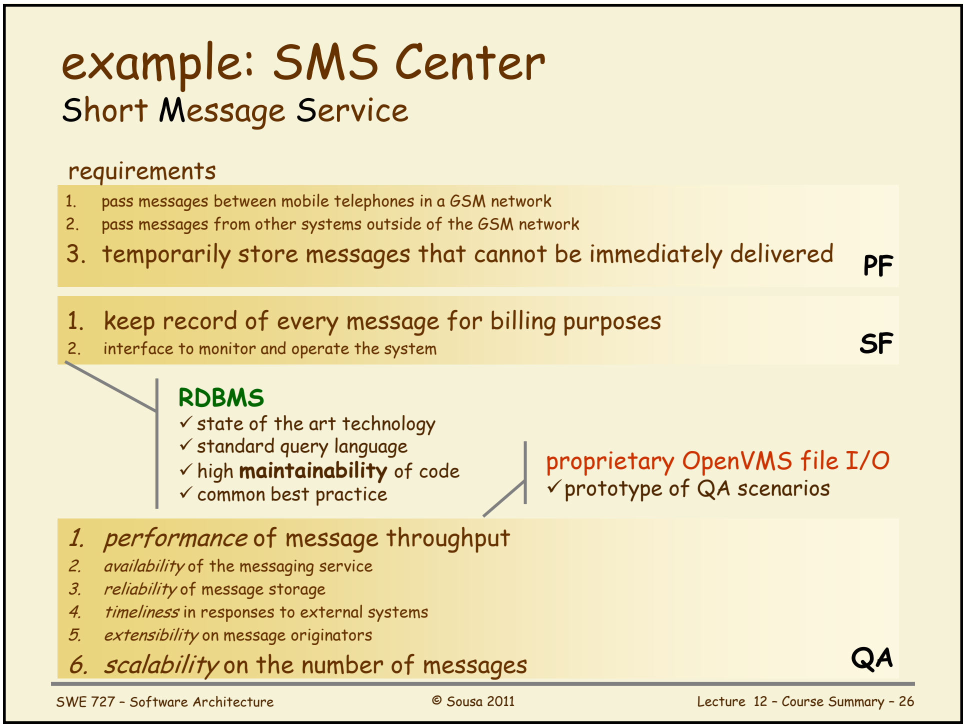
PF

1. keep record of every message for billing purposes
2. interface to monitor and operate the system

SF

1. *performance* of message throughput
2. *availability* of the messaging service
3. *reliability* of message storage
4. *timeliness* in responses to external systems
5. *extensibility* on message originators
6. *scalability* on the number of messages

QA



example: SMS Center

Short Message Service

requirements

1. pass messages between mobile telephones in a GSM network
2. pass messages from other systems outside of the GSM network
3. temporarily store messages that cannot be immediately delivered

PF

1. keep record of every message for billing purposes
2. interface to monitor and operate the system

SF

RDBMS
- ✓ state of the art technology
- ✓ standard query language
- ✓ high **maintainability** of code
- ✓ common best practice

proprietary OpenVMS file I/O
- ✓ prototype of QA scenarios

1. *performance* of message throughput
2. *availability* of the messaging service
3. *reliability* of message storage
4. *timeliness* in responses to external systems
5. *extensibility* on message originators
6. *scalability* on the number of messages

QA

lessons learned

- beware of fashion in system design

1. enumerate all architectural alternatives
2. evaluate each alternative
   relative to the architectural drivers
   - high-level functional requirements
   - constraints
   - quality attributes (QAs)

SWE 727 – Software Architecture © Sousa 2011 Lecture 12 – Course Summary – 27