Software Architecture

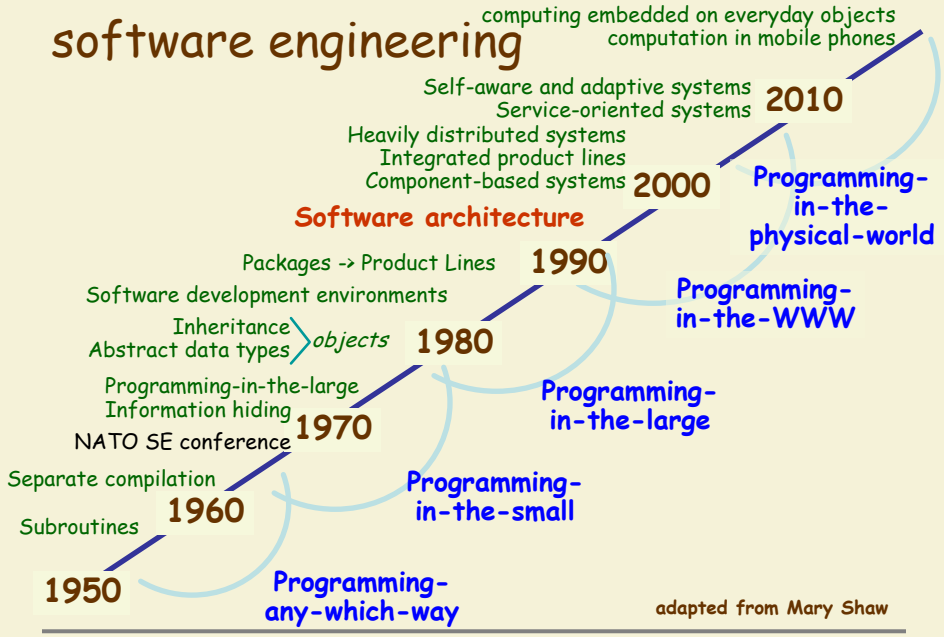
Lecture 10
Course Summary

João Pedro Sousa
George Mason University

outline

- SA in context
  - connectors
- C&C styles
  - data flow
  - call-return
  - events
  - peers
  - service-oriented
- role of SA in
  - addressing change
  - promoting Quality Attributes

software engineering

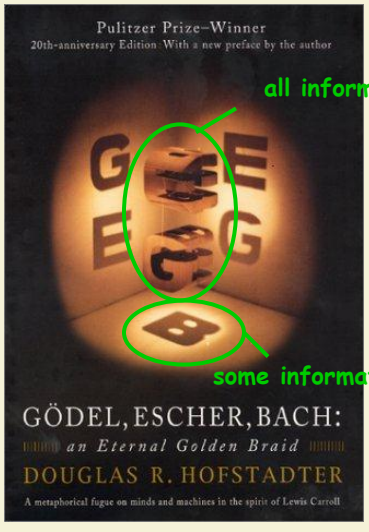


adapted from Mary Shaw

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

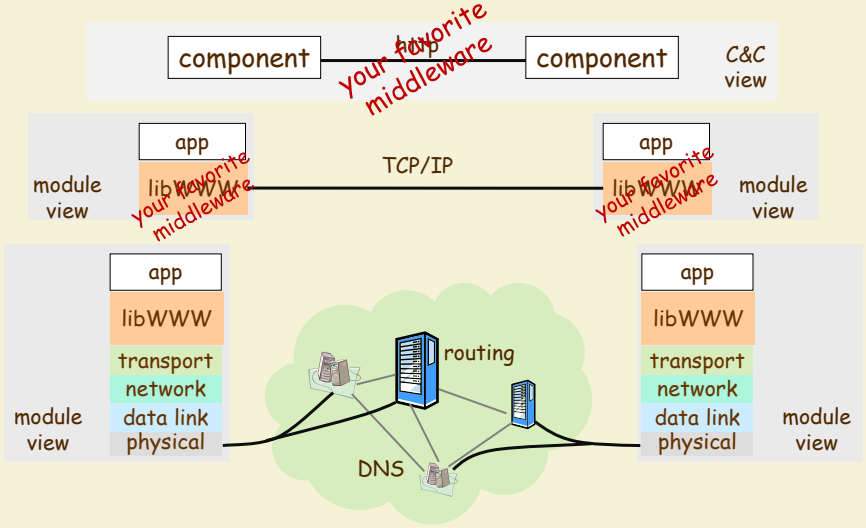
why?



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

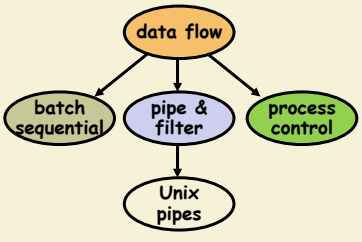
data-oriented repository
- transactional databases
  - stateful client-server
- blackboard
- modern compiler

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



select a data flow style when:

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

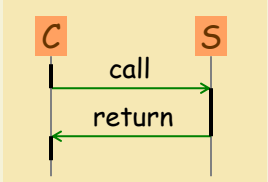
pipe-and-filter architectures are good choices for many data flow applications because

- 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



RPC/RMI

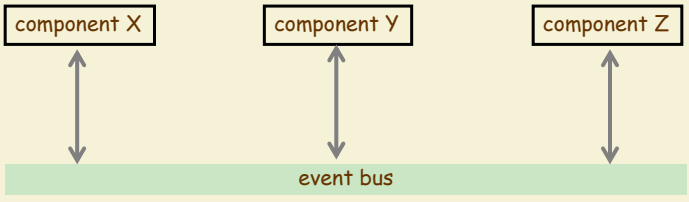
in distributed systems:

- 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

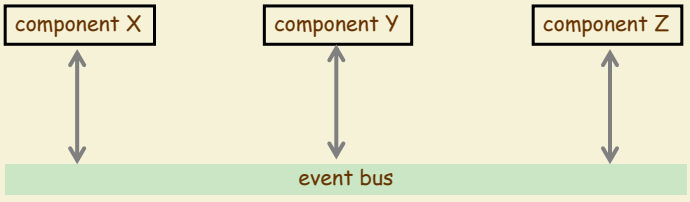
- communicating peers
  - asynchronous messages aka explicit events
    explicit wrt identifying the recipient
- event systems aka implicit events
  - events delivered to all interested components in some order
  - publish aka broadcast
  - publish-subscribe
    - interested components subscribe to events
    - 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

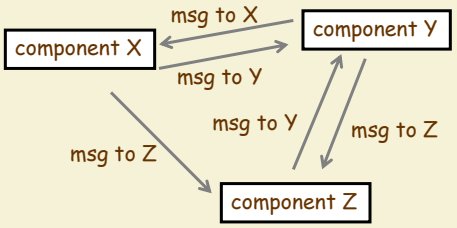


- push / pull
- filtering events in component / bus
- call-return / asynchronous messages
- local / remote comms

discussion
client-server vs. pub-sub

- topology
- distribution
- where data reside
- system wide behavior

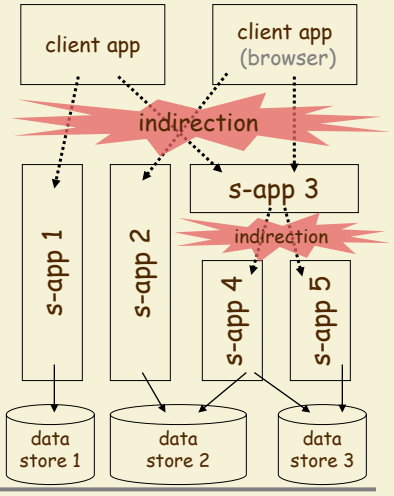
communicating peers
middle ground between call-return & events



	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

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

outline

- SA in context
  - connectors
- C&C styles
  - data flow
  - call-return
  - events
  - peers
  - service-oriented
- role of SA in
  - addressing change
  - promoting Quality Attributes

change
comes with the software *territory*

- reasons for change
  - stakeholders want to
    - customize a new version of entire system
    - add new features to an existing system
    - improve existing features/fix problems
    - remove/restrict access to features
    - improve some QAs
  - application environment changes
    - competition has cool new features
    - platforms/OS/devices evolve
    - resources/load fluctuate during operation

what changes

- code
  - component/connector internal logic
  - component/connector APIs
    - i.e. traditional software maintenance
- parameters of execution/configuration
  - modes of operation in embedded systems
    - e.g. elevators, HVAC, robots, cars…
- C&C run-time structure
  - change connectors while maintaining style
    - e.g. local -> distributed system
  - change features
    - e.g. change service coordination to invoke new kinds of services
  - same features, tune QAs
    - e.g. remapping of service providers, # of replicas of a web server…

architecture models
help with changes

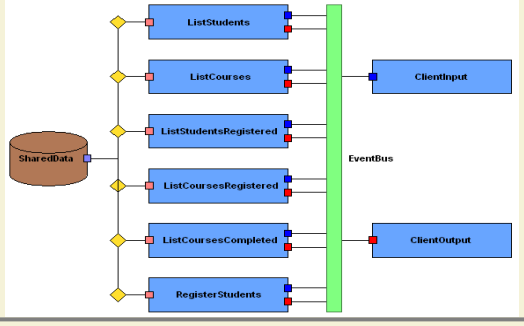
- code
  - offline, by human hands
- parameters of execution/configuration
  - decision made by
    - humans
    - automatically
  - changes effected automatically
- C&C run-time structure
  - change features/architectural constraints
    - offline, by human hands
  - same features, tune QAs
    - offline, by human hands
    - automatically

analyze
scope of impact
effects on QAs…

example change architectural constraints
make systems distributed

- data streaming system based on Java pipes
  e.g. lab1

- pub-sub system based on Java Observer-Observable

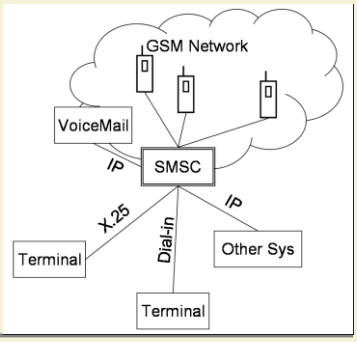


quality is linked to function
non-functional reqs is a misnomer

- 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

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* of response 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)