university of groningen

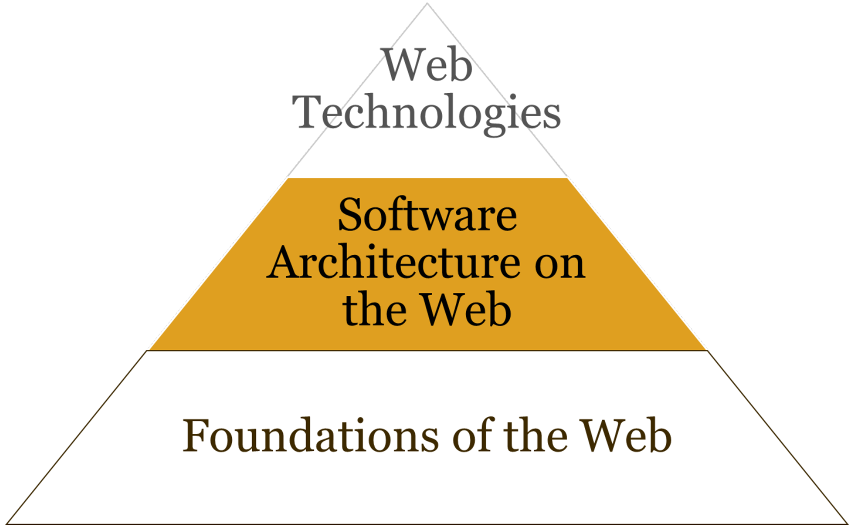
Web Engineering (WBCS008-05)

Set 5: Architectural Concerns

Vasilios Andrikopoulos
v.andrikopoulos@rug.nl

Outline

- Architectural model
- System decomposition
- TBU

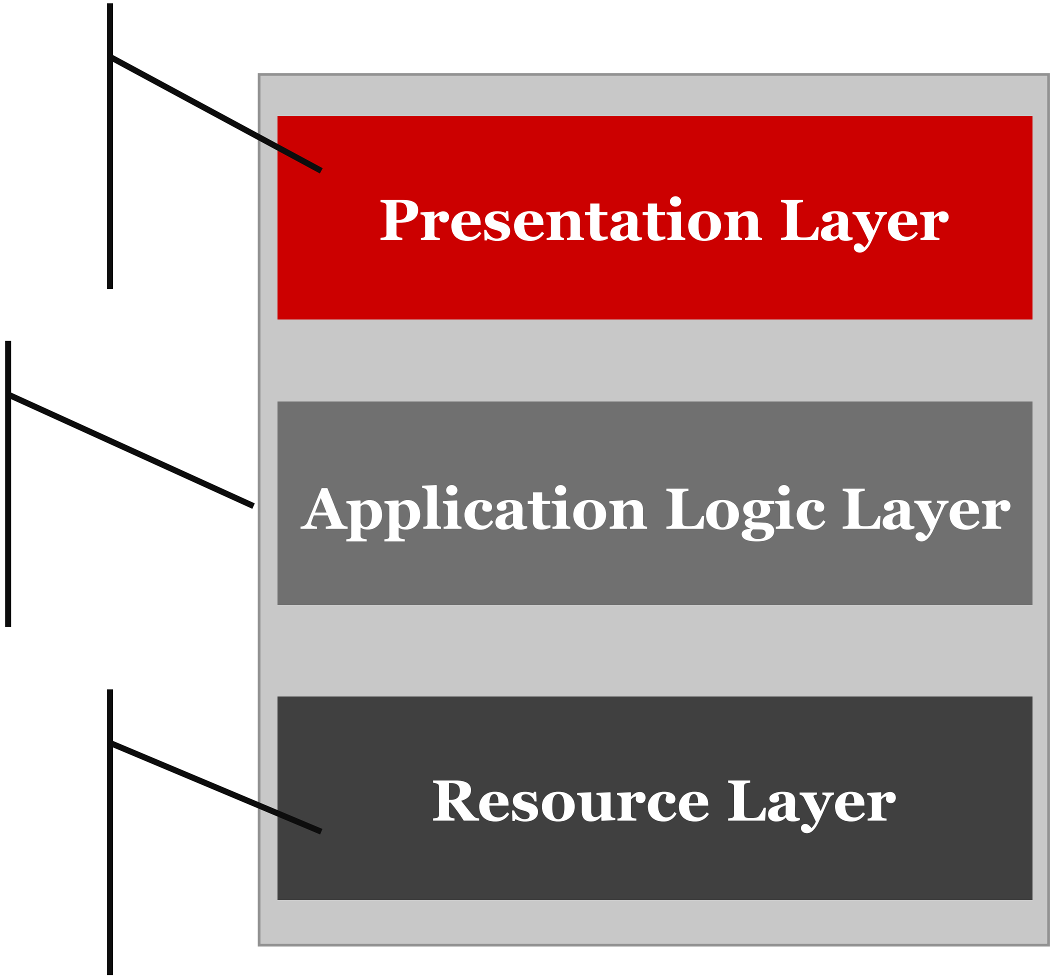


Application Layers (after Fowler)

- Rendering of data
- Reaction to events
- Communication/conversation logic e.g. sessions

- Functions offered via presentation layer
- Logic performed by the application

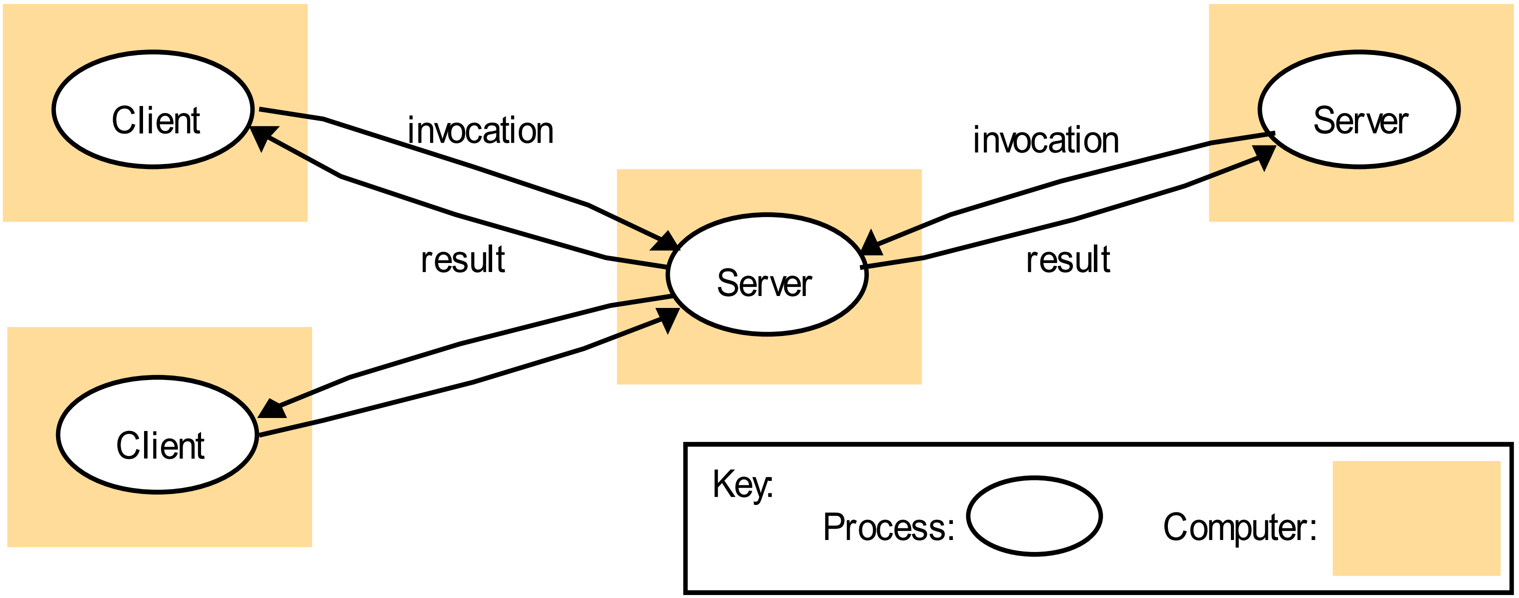
- Logic to access data needed by application logic
- Database, files, content, queues, other support functions



Monolithic applications

- Entirely self-contained in terms of behavior
- Deployed as a single unit
- All-in-one app development
- Simple but difficult to scale operations and development with system size

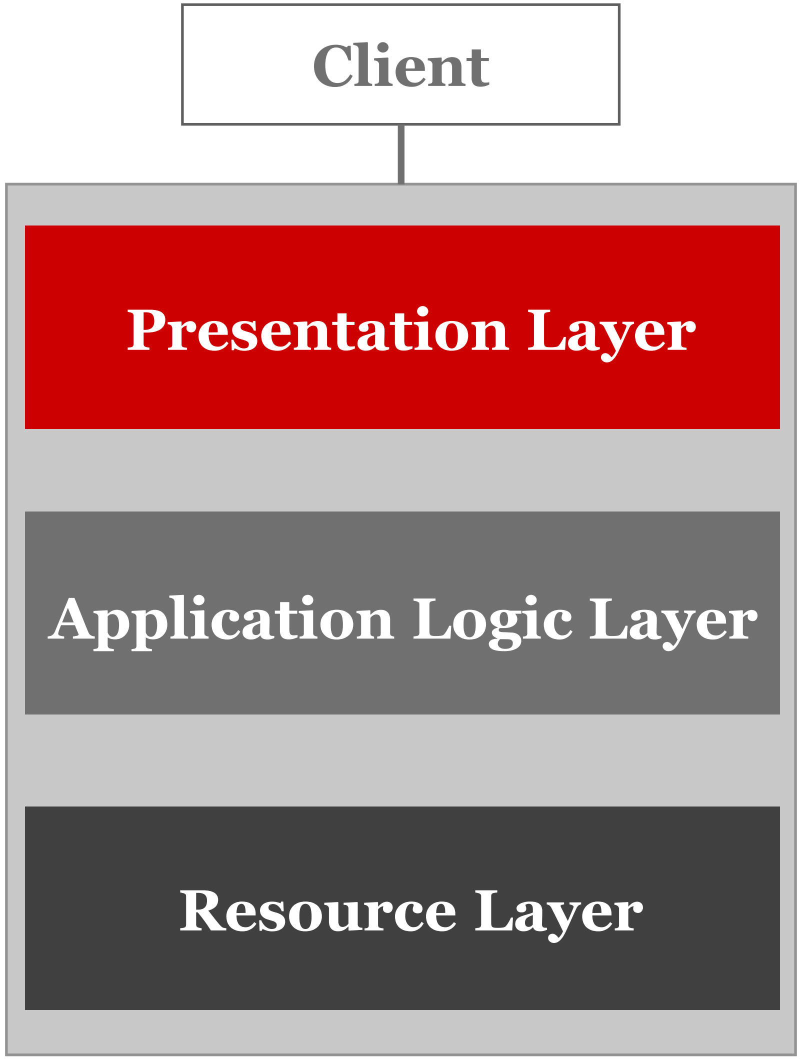
The Client-Server architectural style



From [Coulouris et al.]

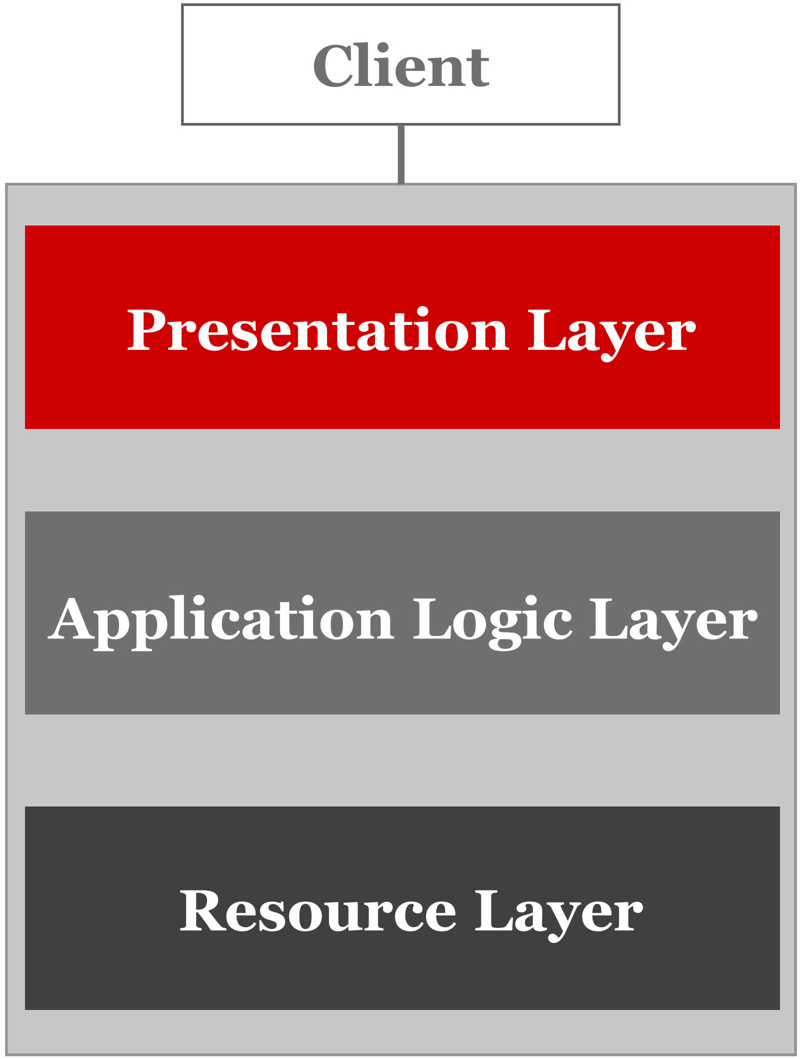
Client vs Presentation Layer

- Thin vs Thick clients defining the relation dynamic
- Thin clients contain little or no presentation logic
  - E.g. browsers without JS
- Thick clients may host parts of the presentation logic (and not only)



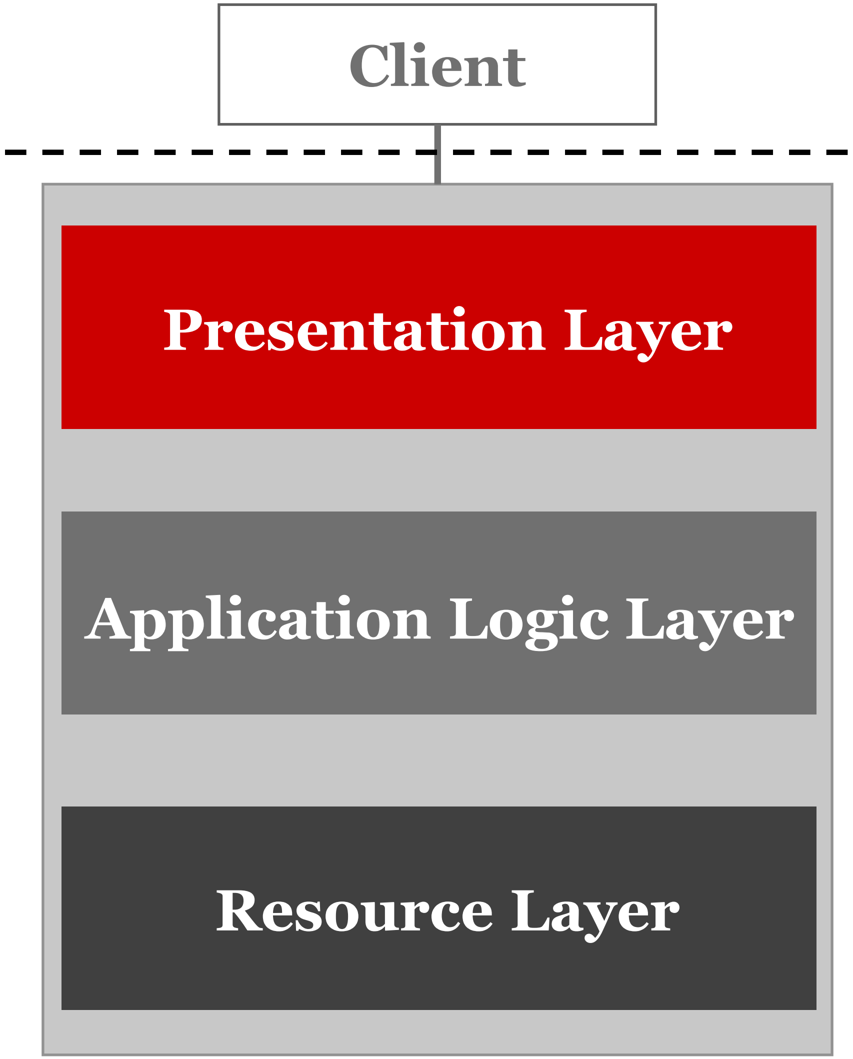
Client-Server x Layers: application splits

- Tiers as physical organization units
  - Client-Server as a 2-tier (at least) model
- Layers as functional organization units
  - Splitting/aggregating layers in one or more tiers results into different topologies
- **Key decision**: how much functionality is "shipped" to the client



Basic 2-tier applications

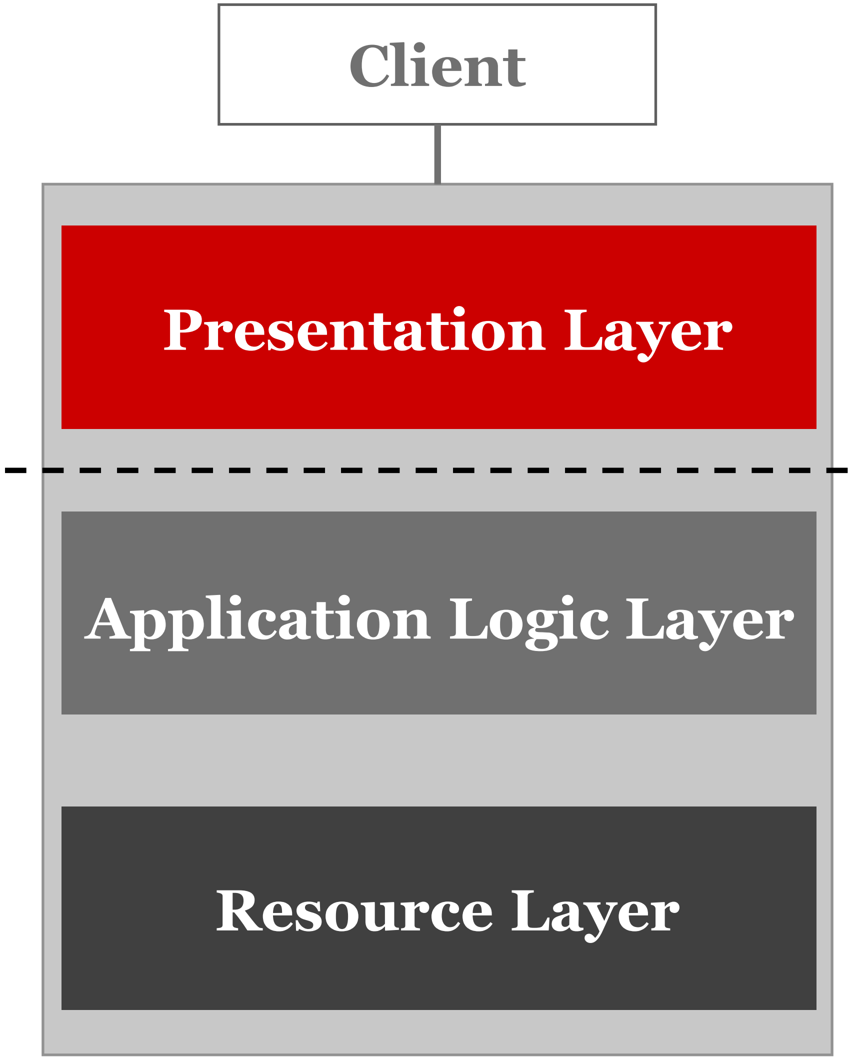
- Thin client model
- Centralized applications with clients acting as dumb terminals
- Effectively a monolithic application
- Example: Static Web sites (HTML only, no JS)



Remote presentation applications

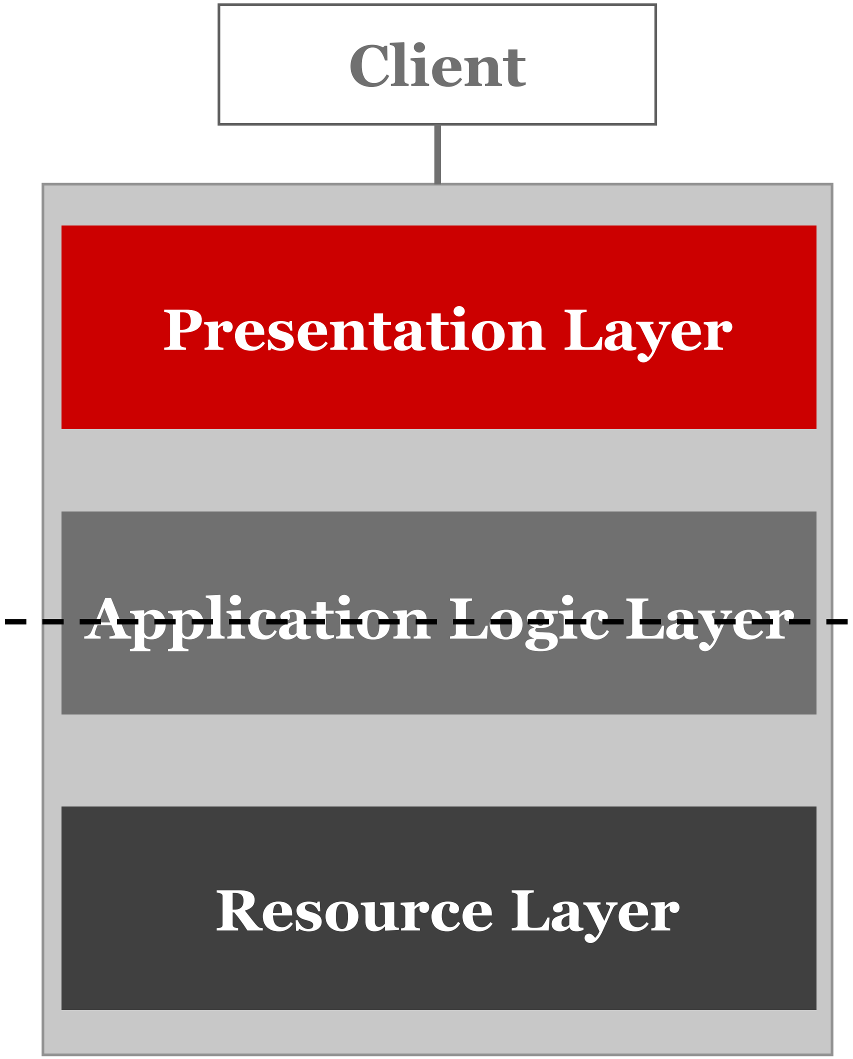
- Thick client model
  - Single Page Applications (SPAs)
  - Multi-device Web applications

- Built on top of APIs



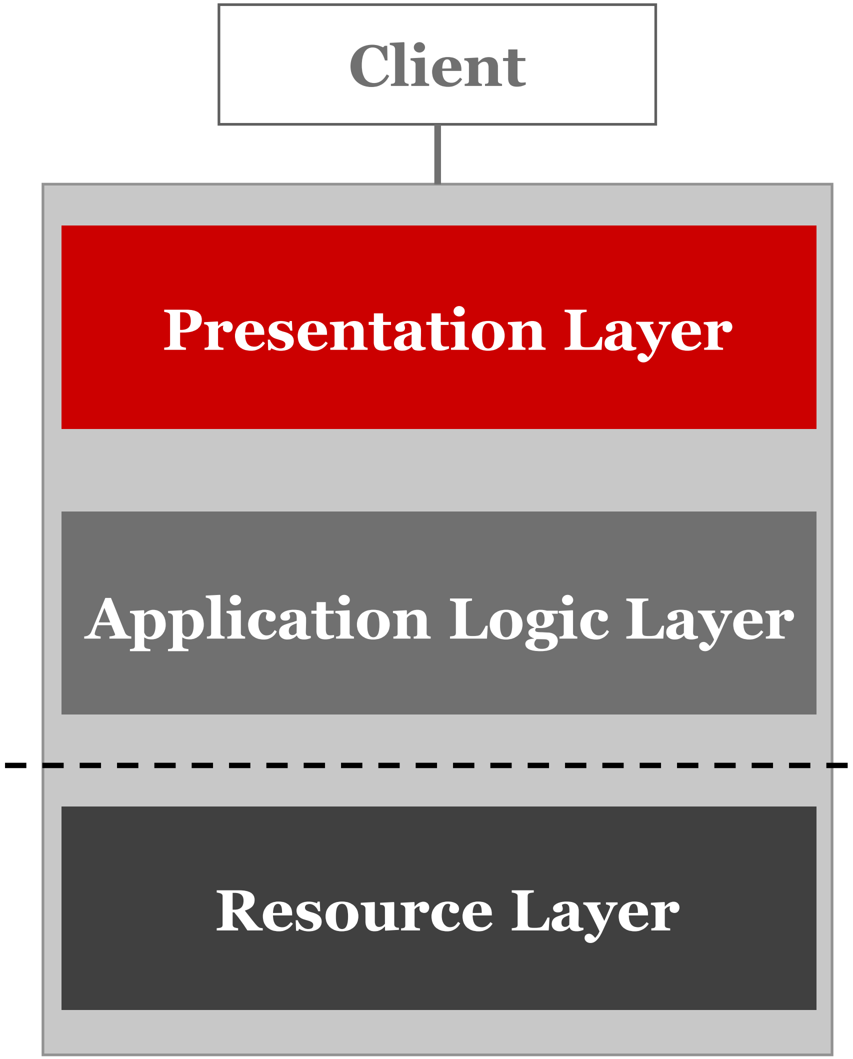
Distributed applications

- Thick client model
  - Single Page Applications (SPAs)
  - Many current Web applications & sites
- Application logic on client delegates (some) processing to function on server



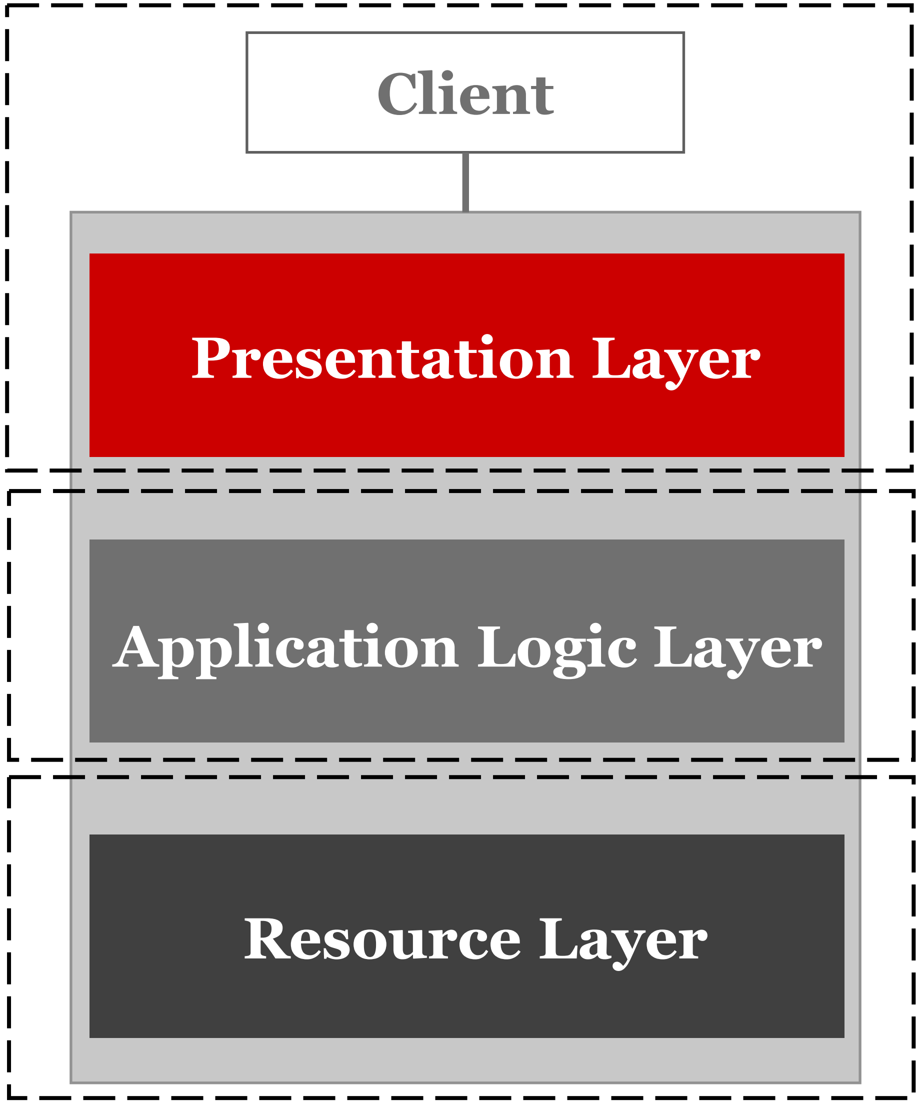
Remote data applications

- Thick client model
  - Mobile/platform games
- Access via high-level interface to resources
- Provides location transparency of resources



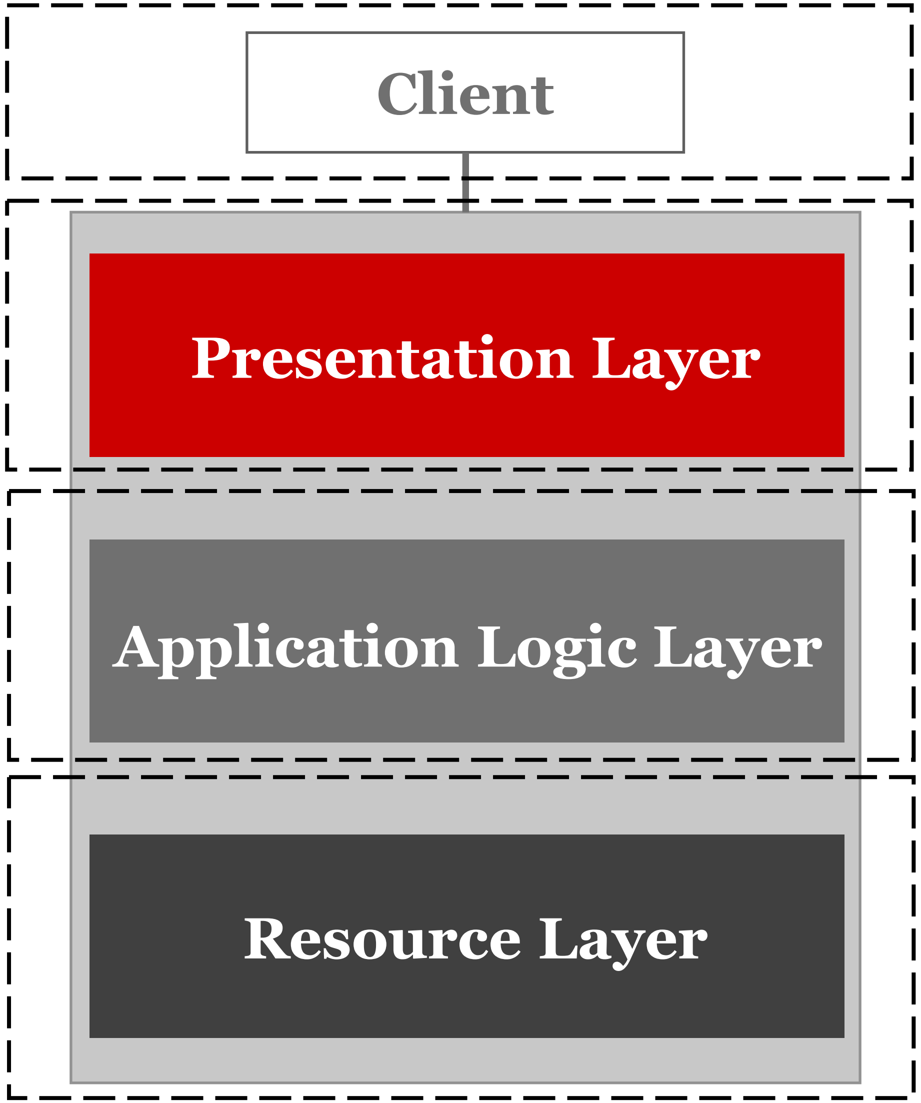
3-tier applications

- Commonly confused with the application layers, e.g. Presentation tier, Logic tier, Data tier
- Tier is meant to denote physical location
- A tier may contain more than one layers or only a part of a layer



Basic n-tier applications

- As many orthogonal cuts to the layers as necessary
- Quite common in large scale Web applications



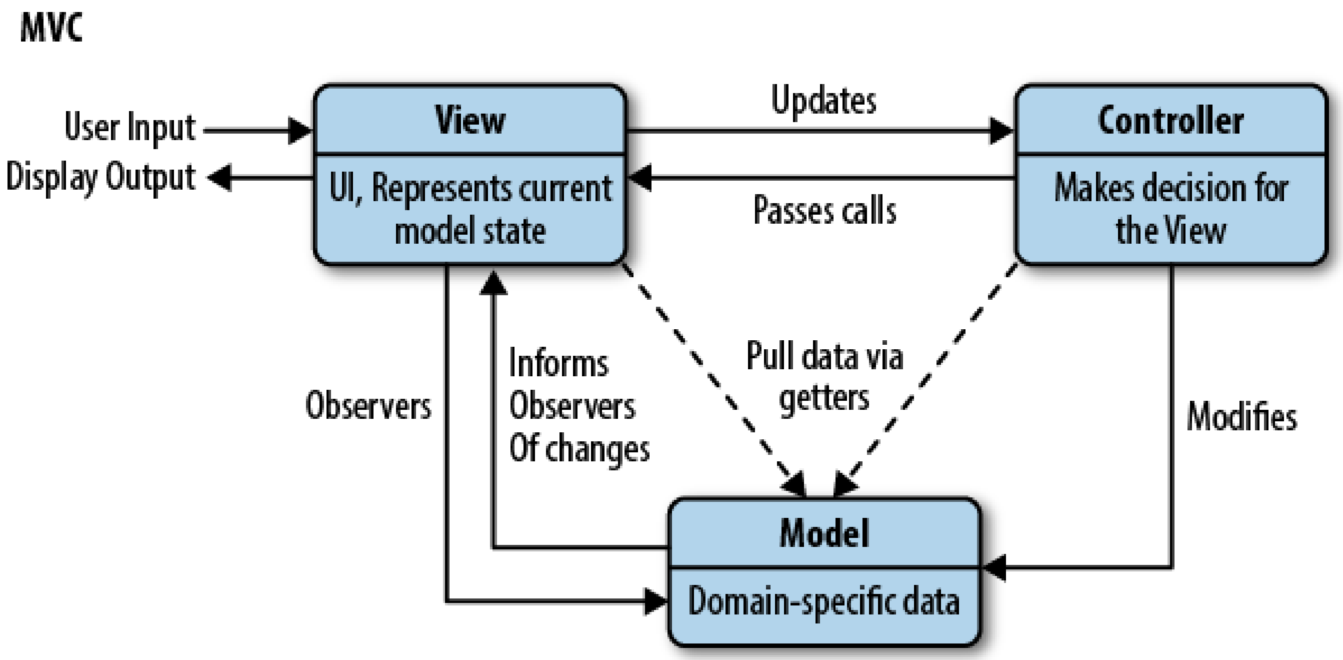
Design Patterns (for JavaScript Web Frameworks)

MV* Patterns

- MVC: Model View Controller (e.g. Django, Angular)
- MVP: Model View Presenter (e.g. Backbone.js)
- MVVM: Model View ViewModel (e.g. Vue)

- MV* Patterns predating JS

MVC x JS



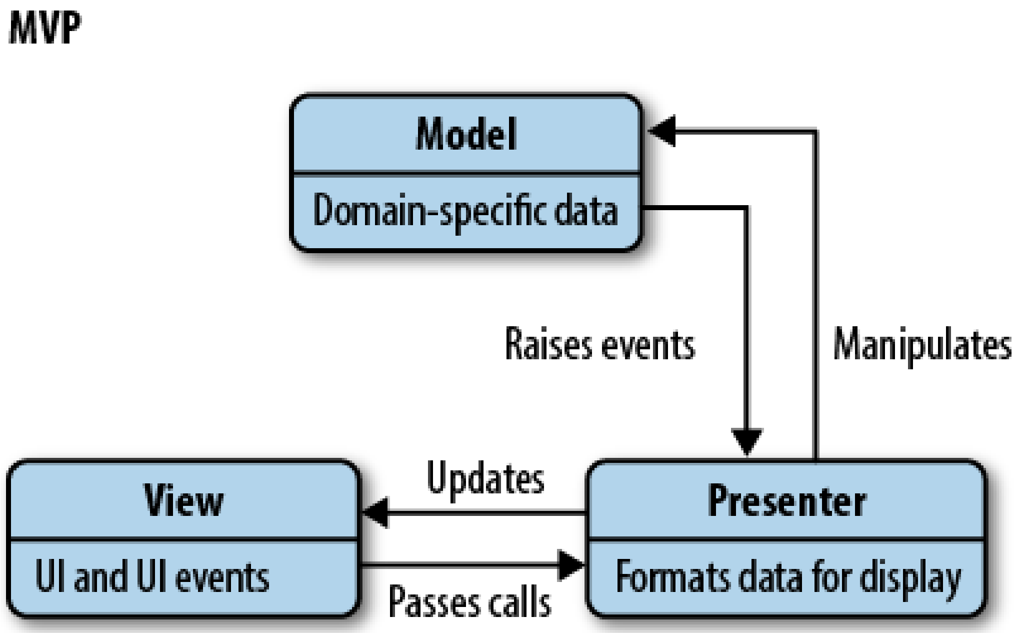
- Many frameworks allow for Model grouping in collections
- Views build and maintain DOM elements
- Templates for Views as HTML markup to avoid over-generating

MVC benefits

- Easier maintenance through separation of concerns
- Easier unit testing due to MV decoupling
- Easier to apply DRY[1] principle due to MV decoupling
- Modularity/separation of concerns allows for parallel development

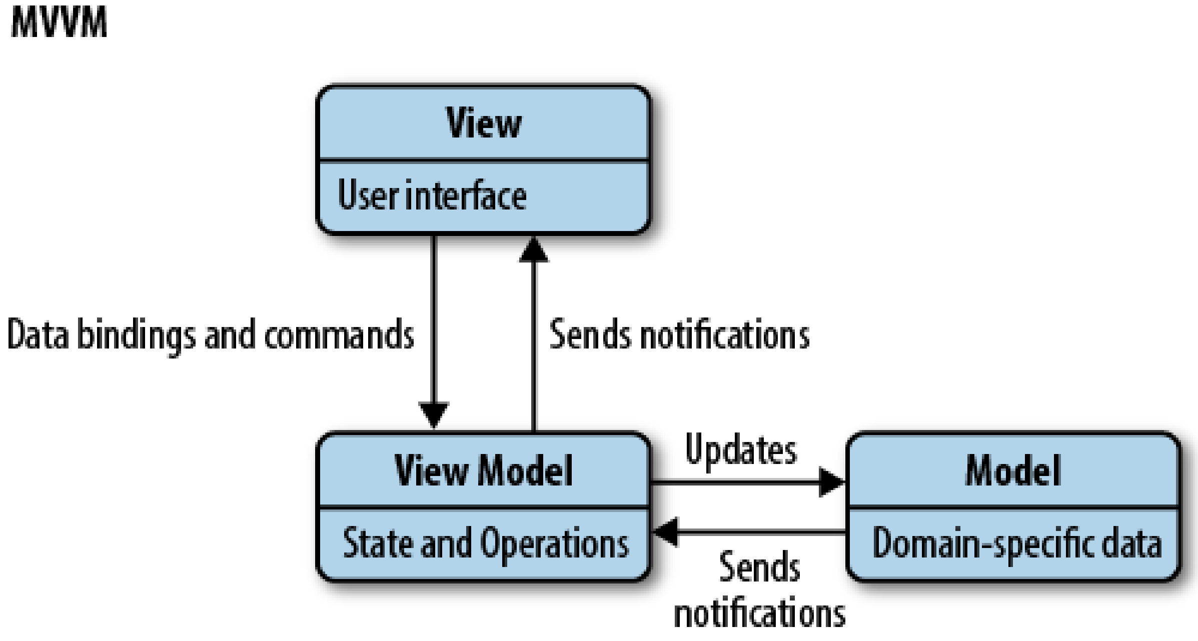
[1] Don’t Repeat Yourself

MVP x JS



- Presenters instead of Controllers
  - Decoupling of VP: communication through interfaces
- Most commonly implemented with a Passive (little to no logic) View
- Increased testability and modularity
- Presentation logic reuse across MVs

MVVM x JS



- ViewModel encapsulating the business logic with View dealing with formatting of data
  - Validation actually done by Models
  - ViewModel as specialized Controller: data converter

MVC vs MVP vs MVVM

- MVP and MVVM as derivatives of MVC
  - Key difference in layers' dependencies/positioning
- MVC: Views on top with direct access to Models
  - Security issues and performance costs
  - Less complexity

MVC vs MVP vs MVVM (continued)

- MVP: Presenters on the same level as Views, mediating between Views and Models
  - Requires Views to implement Presenter interfaces

- MVVM: ViewModels allow for View-specific subsets of Models, not required to reference a View
  - Less logic required for the View
  - Interpretation between VM and V required → Performance costs

Service Oriented Architecture

Service Oriented Architecture (SOA)

- **SOA** is an architectural paradigm (i.e. style) for the realization and maintenance of *business processes* that span large distributed systems [adapted from Josuttis2007]

- Goal: to provide flexibility by supporting *heterogeneity*, *decentralization* and *fault tolerance*

Basic concepts

- Services
  - Services are encapsulated functionality abstractions
  - Exposing clearly defined interfaces
  - Obscuring the “internal” implementation details
  - Many things can be considered a service (even humans…)
- Infrastructure combines these services in an easy and flexible manner
- Policies and processes that deal with the heterogeneity, changeability and multiple ownership of large distributed systems

Basic concepts (cont.)

› Interoperability

- The ability to (easily) connect systems
- As exemplified by Enterprise Application Integration (EAI)

› Loose coupling

- Minimization of dependencies between systems
- Leads to fault-tolerance, flexibility and scalability
- Not binary relation with tight coupling: different degrees of coupling may be present in one system

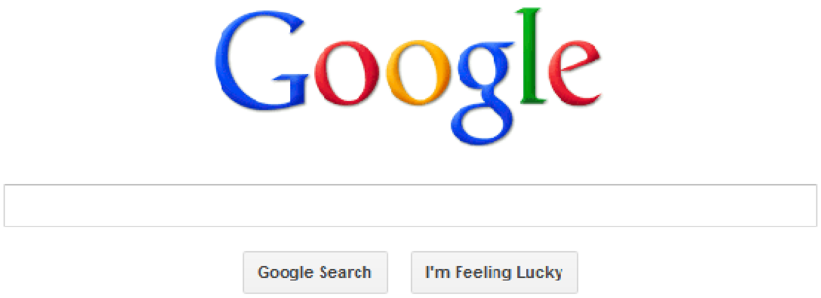
Services constituents

Interface

- Functionality visible to the external world
- Means to access this functionality
- Self-descriptive definition = easy to understand

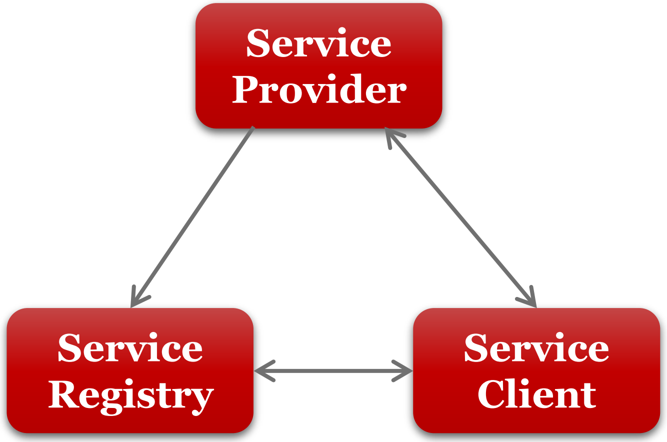
Implementation

- Realizes specific service interface(s)
- Multiple languages/platforms can be used
- May use other services to implement functionality



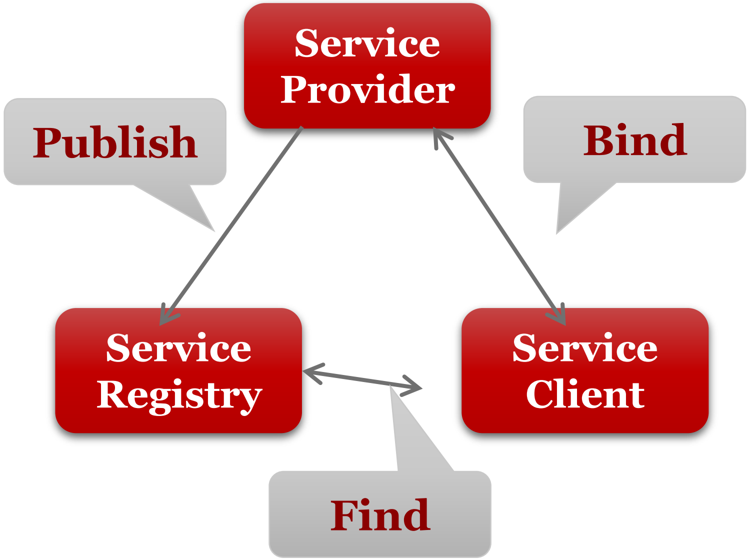
Interaction roles

- **Provider**
    - Organization that owns the service and implements the underlying business logic
    - The platform hosting and controlling access to the service
- **Consumer/Client**
    - An organization requiring certain functionality to be satisfied
    - An application or service that uses the service
- **Registry**
    - Searchable directory where services are described
    - Clients can “discover” suitable services and get all necessary information to use them



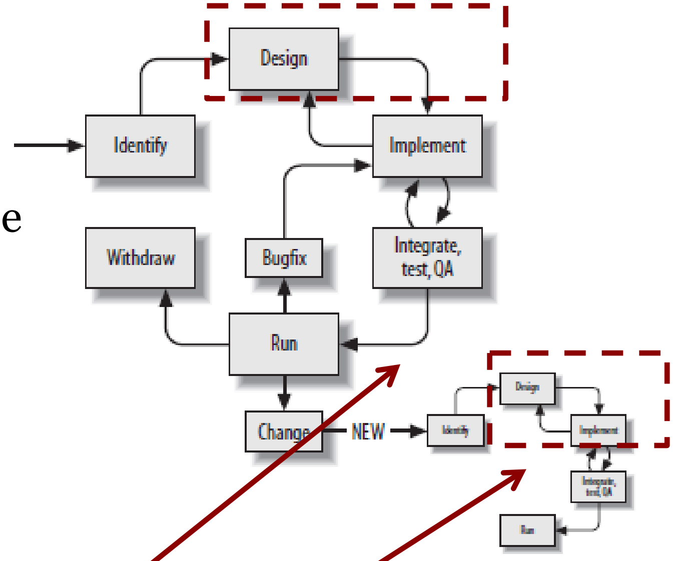
Operations

- › Publish
  - Description
  - Registration
- › Find
  - Discovery
  - Selection
- › Bind
  - Invocation (direct or indirect)

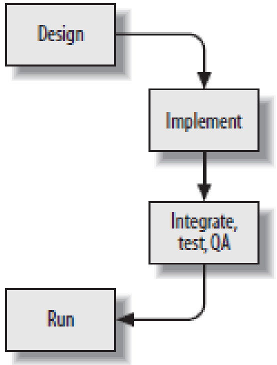


Service Oriented Lifecycle

Lifecycle model for services

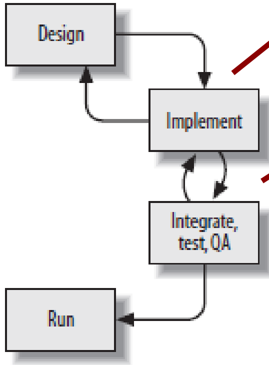


Complete service lifecycle
[Josuttis2007]



Typical SW lifecycle
(the “waterfall” model)

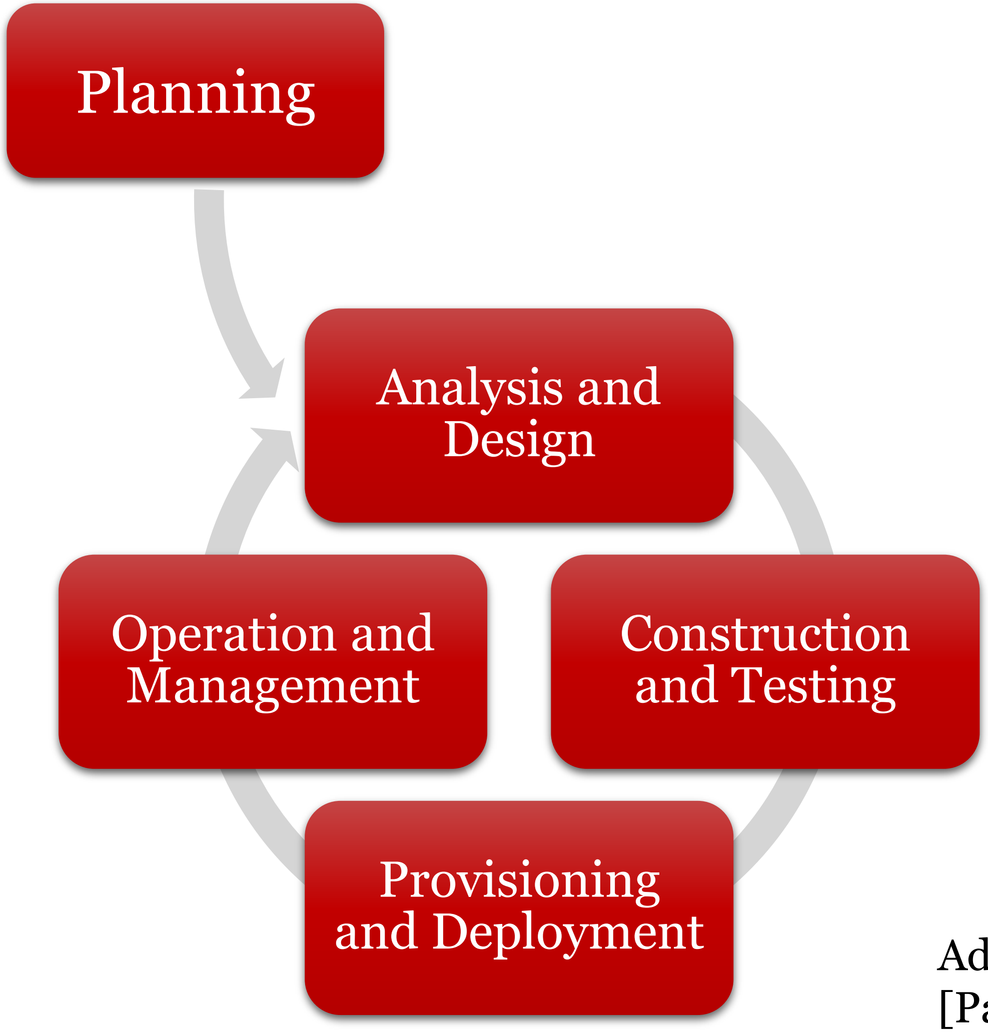
+ iterations



Typical SW lifecycle
with iterations

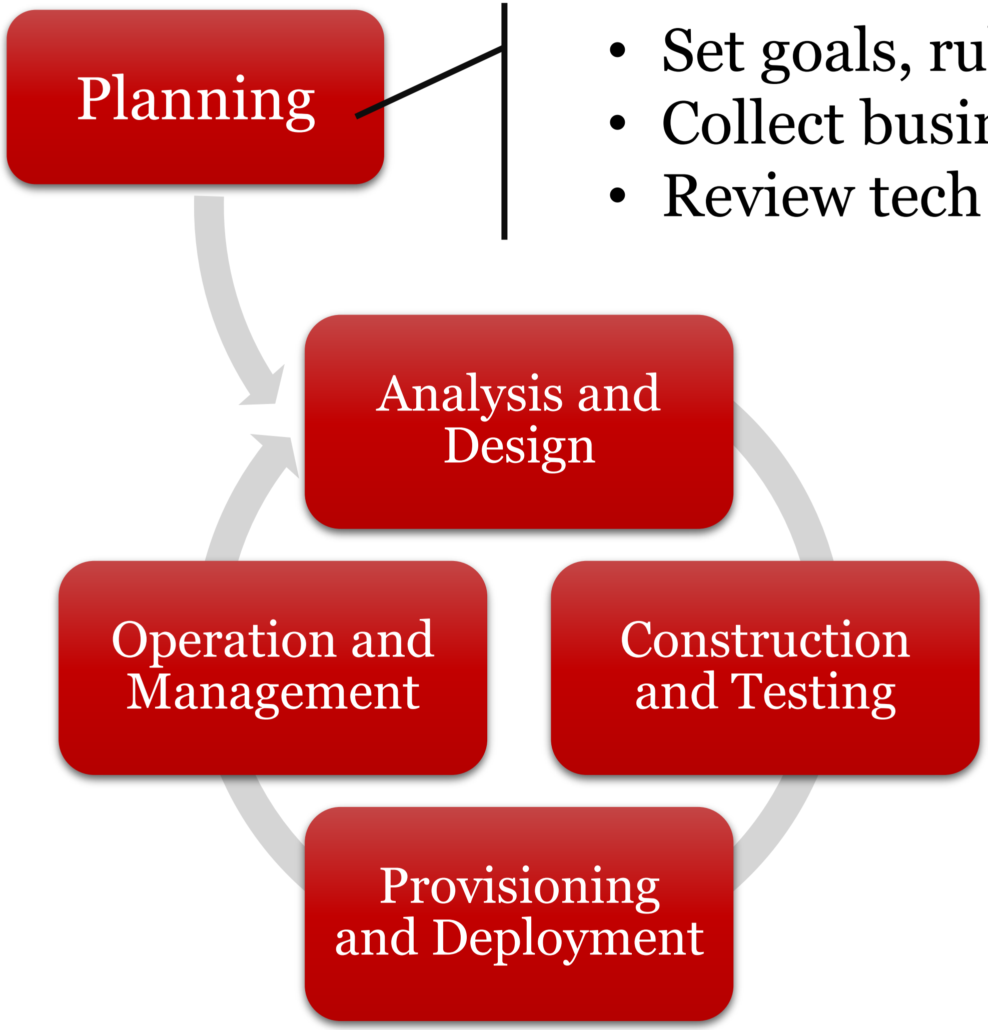
+service identification (FIND)
+modification of running service
+decommission

Reference service lifecycle phases



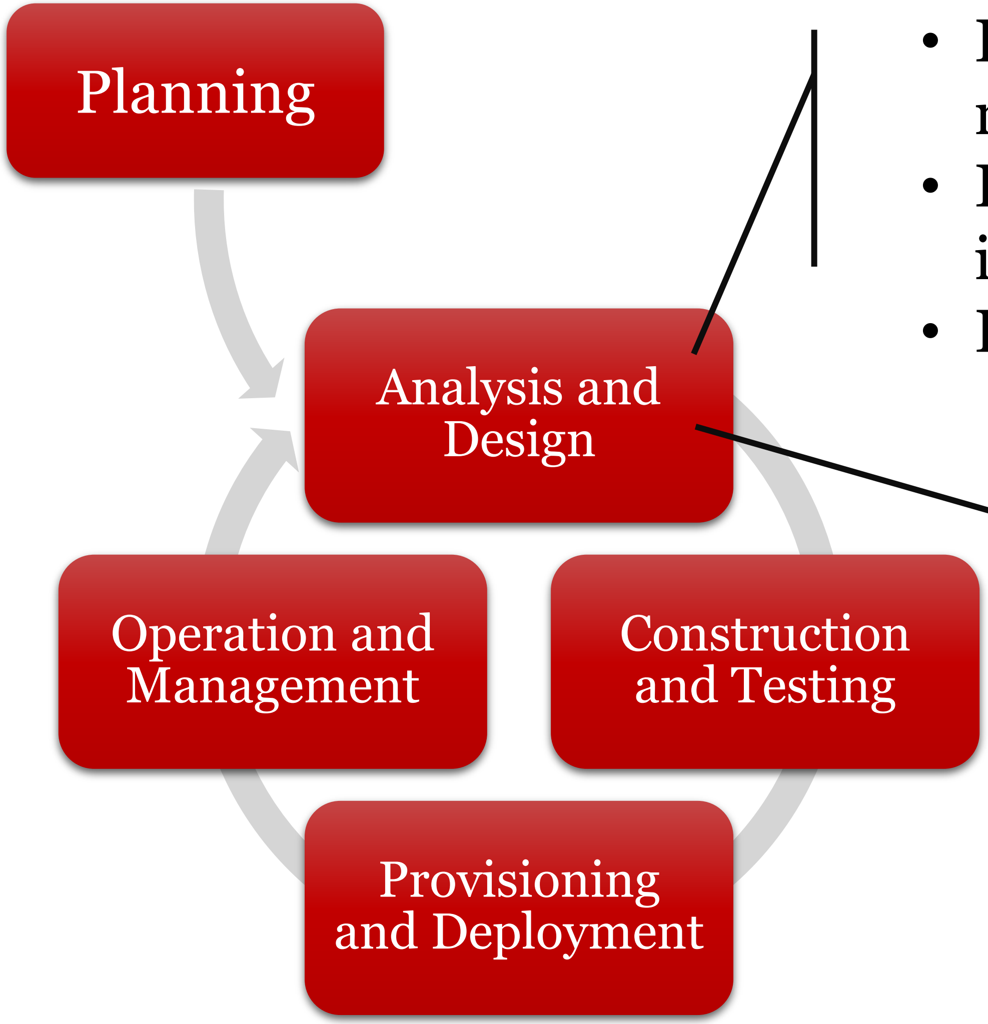
Adapted from [Papazoglou2008]

Reference service lifecycle phases



- Set goals, rules, and procedures
- Collect business requirements
- Review tech landscape

Reference service lifecycle phases



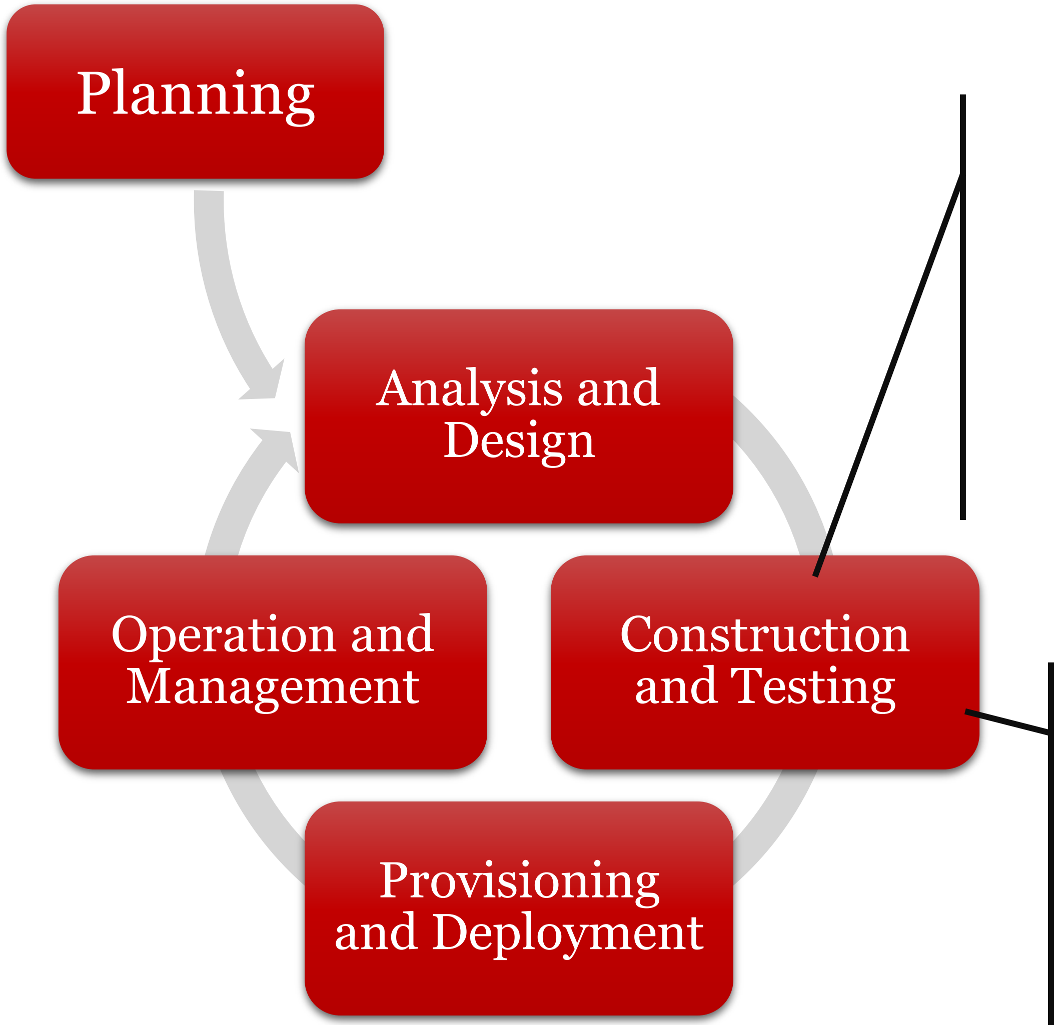
Analysis

- Identify implementation requirements
- Identify business process as set of interacting services
- From “as-is” to “to-be”

Design

- Transform identified processes into service interfaces
- Define granularity
- Focus on reusability and composability of services

Reference service lifecycle phases



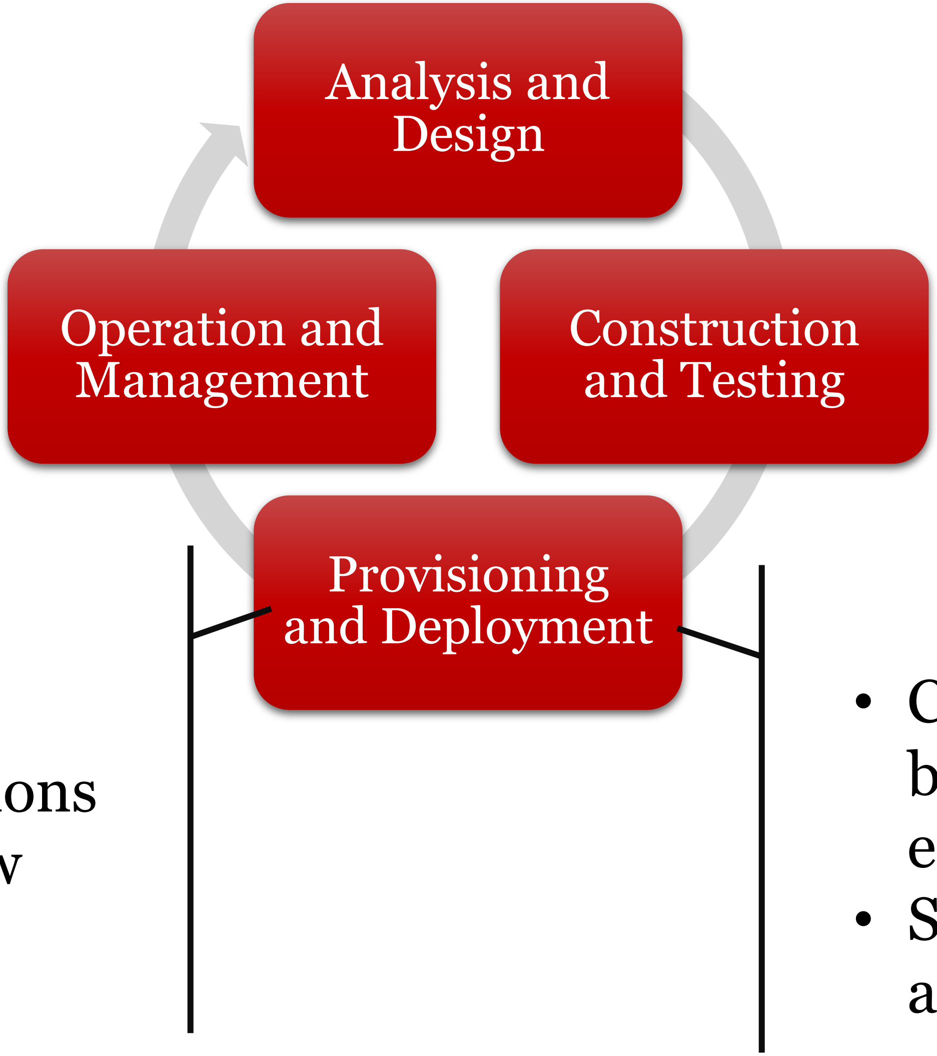
Construction

- Implement specified processes and services
- Create/configure hosting environment
- Assign sufficient resources

Testing

- Analyze/operate services implementation in controlled environment
- Is not necessarily side-effect free

Reference service lifecycle phases



Deployment

- Roll out new services, processes, and applications
- Includes publishing new interfaces in registries

Provisioning

- Combine technical with business aspects in order to ensure operation of services
- Service governance a key aspect

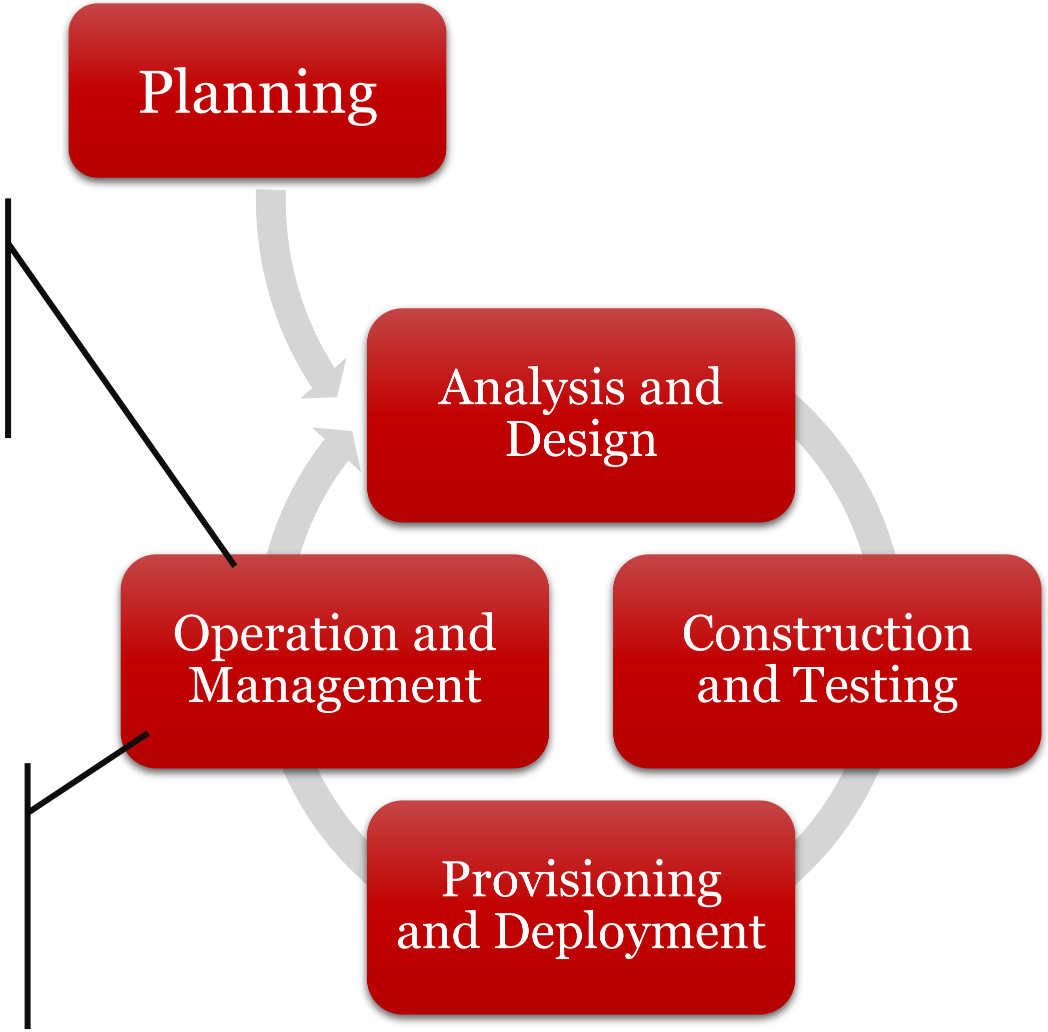
Reference service lifecycle phases

Management

- Monitor Quality of Service (QoS) levels against Service Level Agreements (SLAs)
- Make executive decisions regarding services lifecycle

Operation

- Ensure that services run as planned



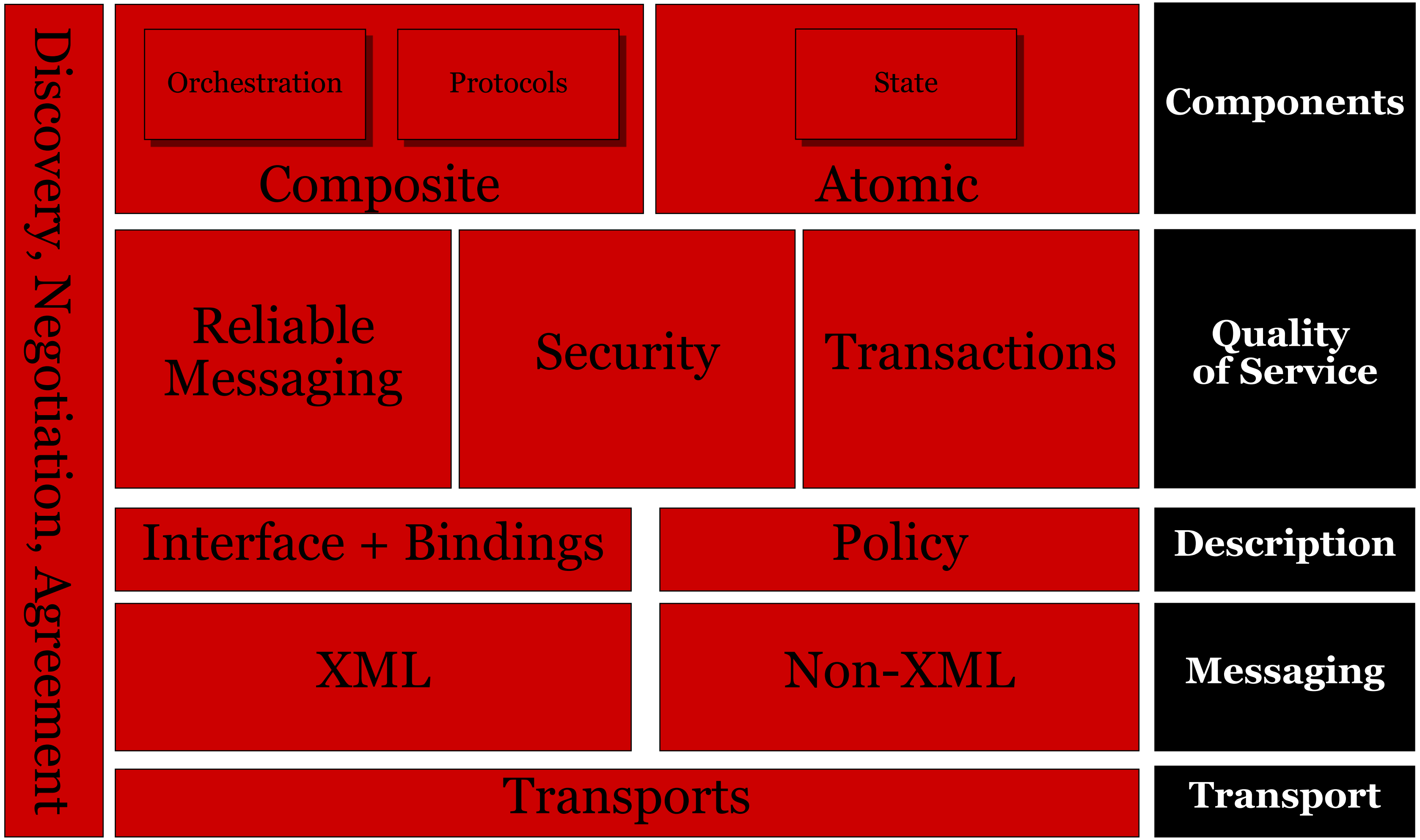
REST vs ~~SOA~~ WS-*

Web Services (WS)

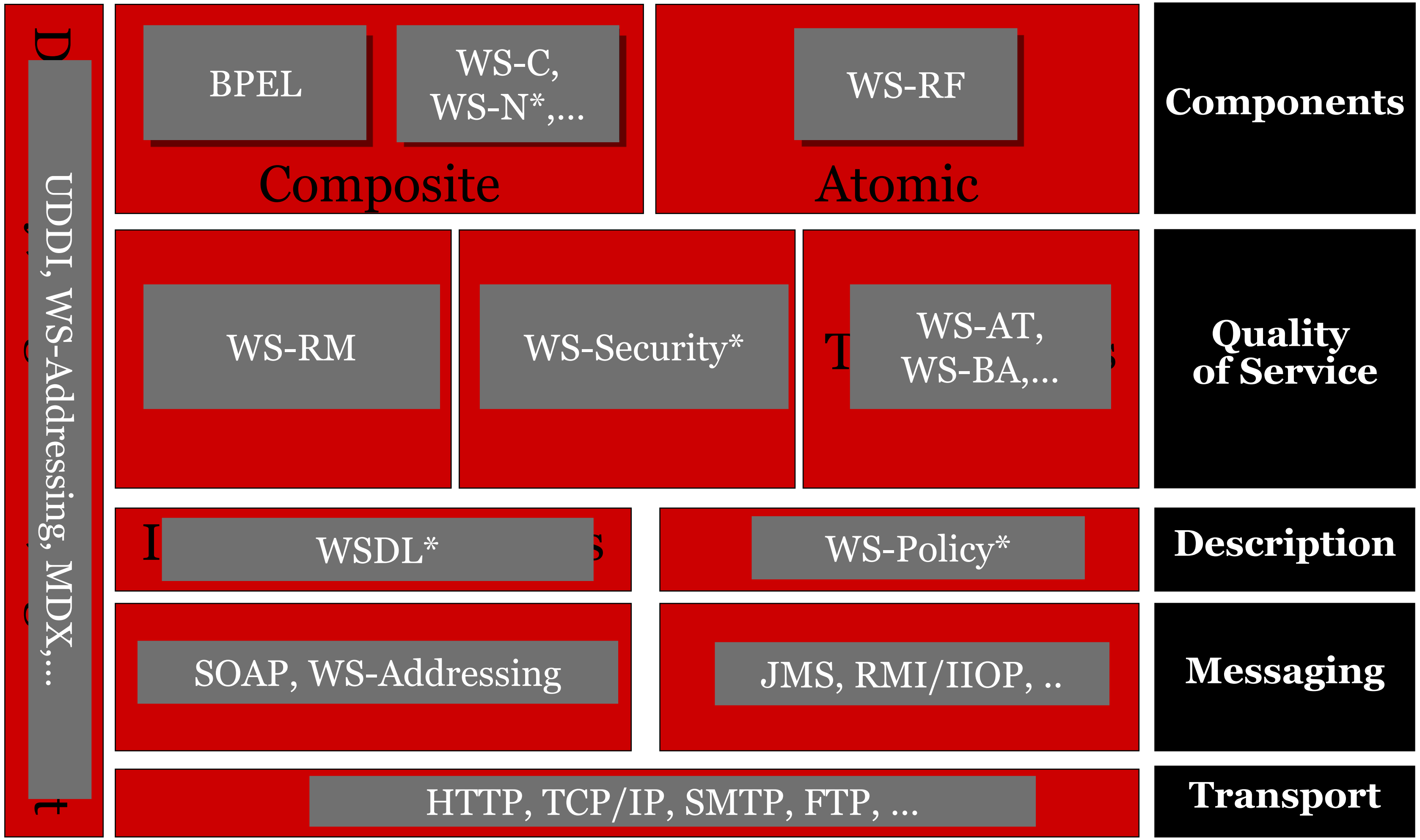
A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Taken from https://www.w3.org/TR/ws-gloss/

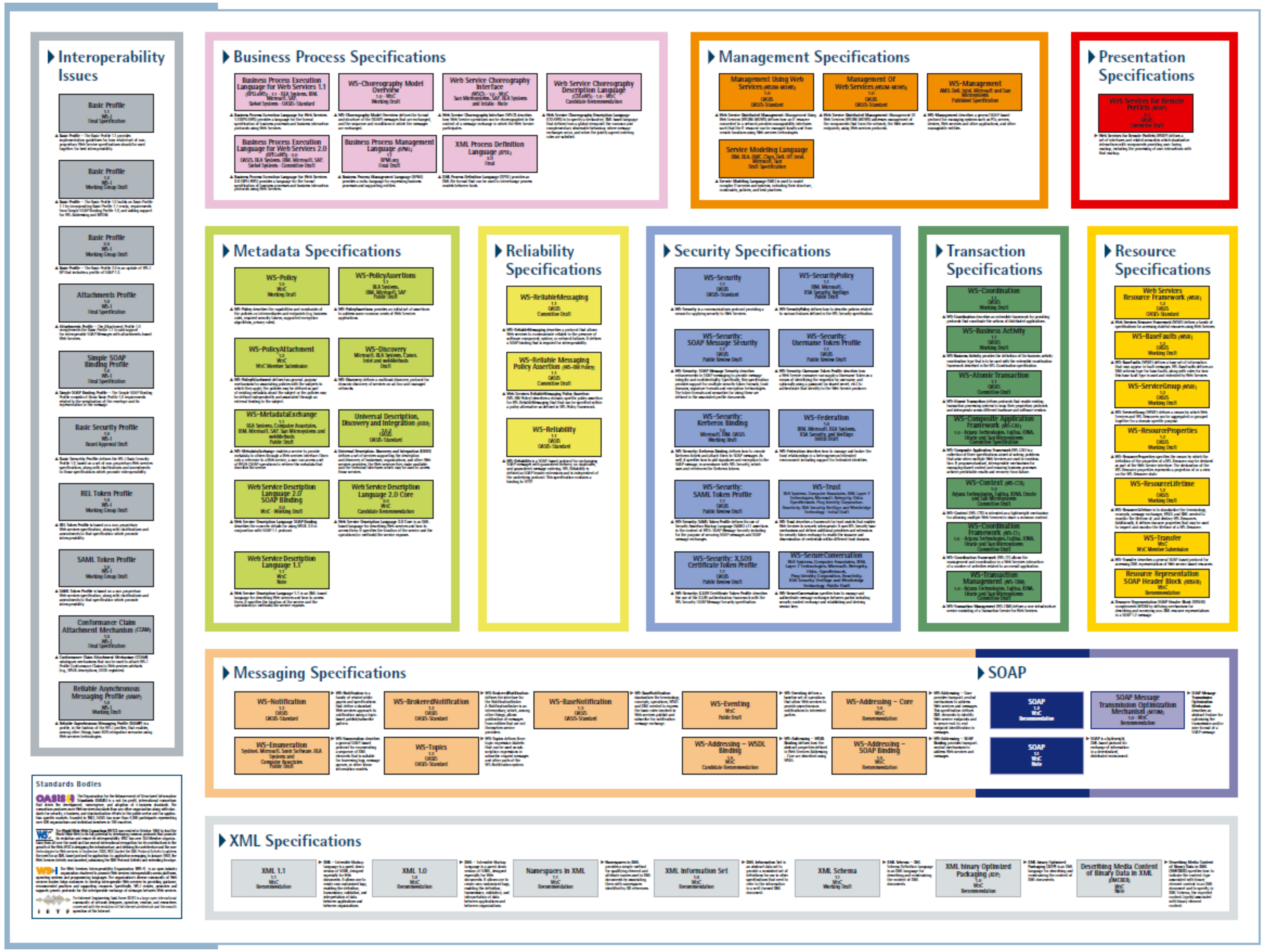
WS-* Stack


Discovery, Negotiation, Agreement
Orchestration
Protocols
Composite
State
Atomic
Components
Reliable Messaging
Security
Transactions
Quality of Service
Interface + Bindings
Policy
Description
XML
Non-XML
Messaging
Transports
Transport

WS-* Stack


BPEL
WS-C,
WS-N*,…
Composite
WS-RF
Atomic
Components
UDDI, WS-Addressing, MDX,…
WS-RM
WS-Security*
WS-AT,
WS-BA,…
Quality
of Service
WSDL*
WS-Policy*
Description
SOAP, WS-Addressing
JMS, RMI/IIOP, ..
Messaging
HTTP, TCP/IP, SMTP, FTP, …
Transport

WS-* Stack (pre-extinction)


Interoperability Issues
Basic Profile
Basic Profile
Basic Profile
Attachments Profile
Simple SOAP Binding Profile
Basic Security Profile
REL Token Profile
SAML Token Profile
Conformance Claim Attachment Mechanism (CCAM)
Reliable Asynchronous Messaging Profile (RAMP)
Standards Bodies
OASIS
W3C
WS-I
IETF
Business Process Specifications
Business Process Execution Language for Web Services 1.1
WS-Choreography Model Overview
Web Service Choreography Interface
Web Service Choreography Description Language
Business Process Execution Language for Web Services 2.0
Business Process Management Language (BPML)
XML Process Definition Language (XPDL)
Management Specifications
Management Using Web Services (WSDM-MUWS)
Management Of Web Services (WSDM-MOWS)
WS-Management
Service Modeling Language
Presentation Specifications
Web Services for Remote Portlets (WSRP)
Metadata Specifications
WS-Policy
WS-PolicyAssertions
WS-PolicyAttachment
WS-Discovery
WS-MetadataExchange
Universal Description, Discovery and Integration (UDDI)
Web Service Description Language 2.0 SOAP Binding
Web Service Description Language 2.0 Core
Web Service Description Language 1.1
Reliability Specifications
WS-ReliableMessaging
WS-Reliable Messaging Policy Assertion (WS-RM Policy)
WS-Reliability
Security Specifications
WS-Security
WS-SecurityPolicy
WS-Security: SOAP Message Security
WS-Security: Username Token Profile
WS-Security: Kerberos Binding
WS-Federation
WS-Security: SAML Token Profile
WS-Trust
WS-Security: X.509 Certificate Token Profile
WS-SecureConversation
Transaction Specifications
WS-Coordination
WS-Business Activity
WS-Atomic Transaction
WS-Composite Application Framework (WS-CAF)
WS-Context (WS-CTX)
WS-Coordination Framework (WS-CF)
WS-Transaction Management (WS-TXM)
Resource Specifications
Web Services Resource Framework (WSRF)
WS-BaseFaults (WSRF)
WS-ServiceGroup (WSRF)
WS-ResourceProperties
WS-ResourceLifetime
WS-Transfer
Resource Representation SOAP Header Block (RRSHB)
Messaging Specifications
WS-Notification
WS-BrokeredNotification
WS-BaseNotification
WS-Eventing
WS-Addressing – Core
WS-Enumeration
WS-Topics
WS-Addressing – WSDL Binding
WS-Addressing – SOAP Binding
SOAP
SOAP
SOAP Message Transmission Optimization Mechanism (MTOM)
SOAP
XML Specifications
XML 1.1
XML 1.0
Namespaces in XML
XML Information Set
XML Schema
XML binary Optimized Packaging (XOP)
Describing Media Content of Binary Data in XML (DMCBDX)

WS-* vs RESTful: Strengths

WS-*

- Protocol transparency and independence (SOAP over HTTP/SMTP/etc.)
- Machine-readable service interfaces defining both the syntax and the semantics of exchanged messages
- Support both synchronous and asynchronous communication
- Complexity hiding behind interface definition
- Interoperability delegated to runtime environment and tool specification conformance

RESTful

- Simplicity through conformance to well-known standards
- Pervasive support infrastructure (HTTP servers and clients)
- Lightweight infrastructure with minimal tooling allows for fast adoption and deployment of new services
- No dedicated registry is required for resource discovery
- Scaling a resource is relatively straightforward through caching, clustering and load balancing
- Lightweight message formats like JSON allow for better performance

WS-* vs RESTful: Weaknesses

WS-*

› Leakage across abstraction layers is common when existing components are transformed into services

› Interoperability issues also arise in the implementation of the WS-* standards, especially the earlier ones

› Translation between XML and object-oriented language constructs leads to performance inefficiencies

RESTful

› Many proxy servers and firewalls allow only POST and GET, requiring non-standard workarounds

› Safe requests (using only GET) having large amounts of input data are often impossible to encode in the URI

› No commonly accepted mechanism for the marshalling of complex data structures

Source material

- Fowler, Martin. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co. Inc., 2002.
- Coulouris, George F., Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. Pearson Education, 2005.
- Addy Osmani’s Learning JavaScript Design Patterns
- Josuttis, Nicolai M. *SOA in practice: the art of distributed system design*. O'Reilly Media, Inc., 2007.
- Papazoglou, Michael. *Web services: principles and technology*. Pearson Education, 2008.
- Pautasso, Cesare, Olaf Zimmermann, and Frank Leymann. "RESTful Web Services vs. “Big” Web Services: Making the Right Architectural Decision." In *Proceedings of the 17th international conference on World Wide Web*, pp. 805-814. ACM, 2008.

Self-evaluation questions

- What are the functionalities encapsulated in each of the application layers in the respective pattern by Fowler?
- What is the relation between layers and tiers? Which architectural decision is important for this decision?
- Where does the application split lie in the case of remote presentation/distributed/remote data applications? Name examples of such types of applications
- How is the MVX pattern defined for JavaScript-using Web applications, where X= {C, P, VM}?

Self-evaluation questions

- What are the main constituents of a service in SOA, and what is their purpose?
- What are the main interaction roles in the SOA triangle, and what operations are they supporting?

Next lecture(s)

Tutorials