Vrije Universiteit Brussel

Advanced Software Development Models and Frameworks CBSD - SOA - AOSD

Viviane Jonckers

2012

Middleware and Components

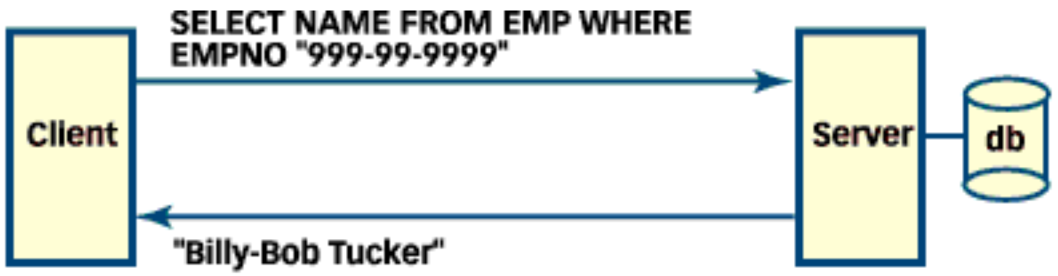
Client Server History

- Information and processing centralized on dedicated machines
  - 1960s mainframe computer
  - 1970s minicomputers
- Information and processing bundled but scattered
  - 1980s personal computers: scattered information, inconsistency and data loss
- Information centralised, widespread data access
  - client/server architectures

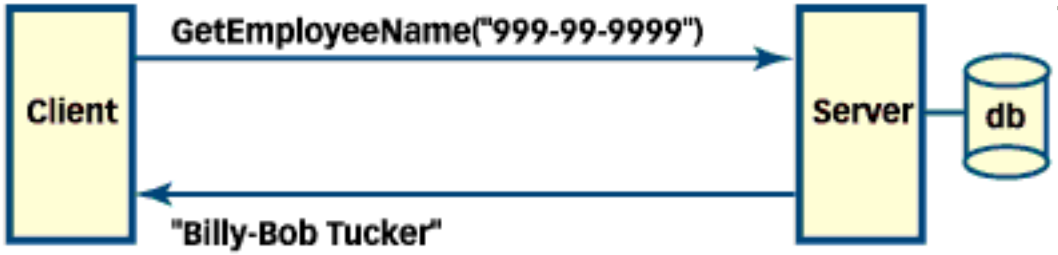
Vrije Universiteit Brussel

Two-tier systems

- Database server:
  - client/server communication through a database language
  - business processes on client, server for data persistency and integrity



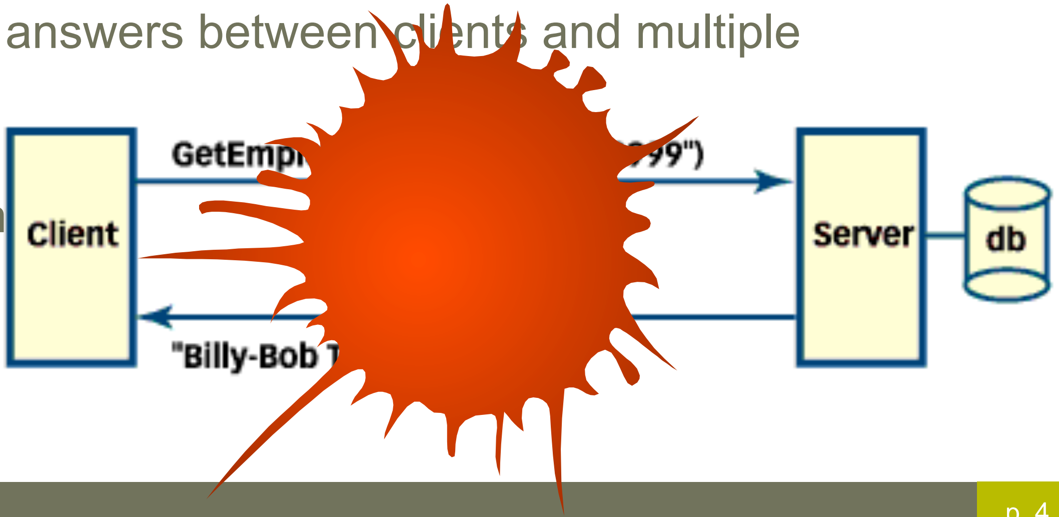
- Application server:
  - clients/server communication on the level of a business transaction
  - business processes on server, clients request execution



Vrije Universiteit Brussel

Middleware (More-tier systems)

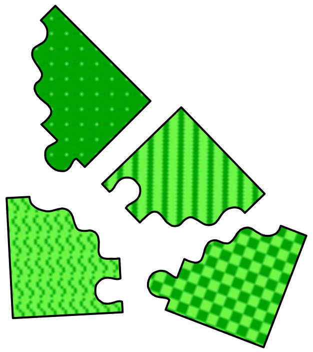
- Pops up to provide flexibility and interoperability
- Addresses some of the liabilities of 2-tier systems
- Examples:
  - transaction processing monitor:
    - streams of requests from multiple clients
    - load balancing between different servers
    - failover on server fails
  - communication protocol translation
  - consolidate requests and answers between clients and multiple heterogeneous servers
  - service metering
  - network traffic information
  - …

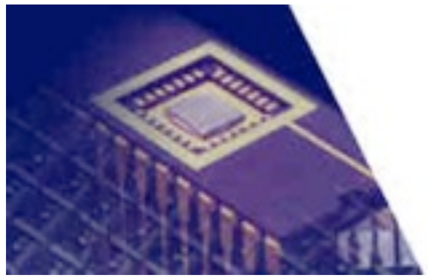


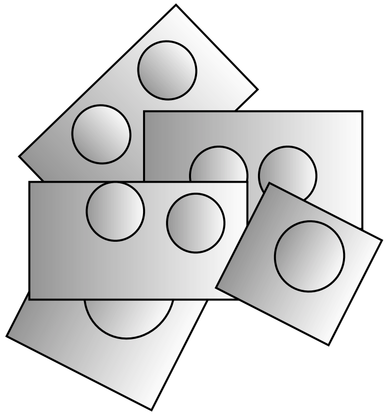
Software Components

" A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties "

Formulated at the 1996 ECOOP conference (Szyperski and Pfister, 1997)





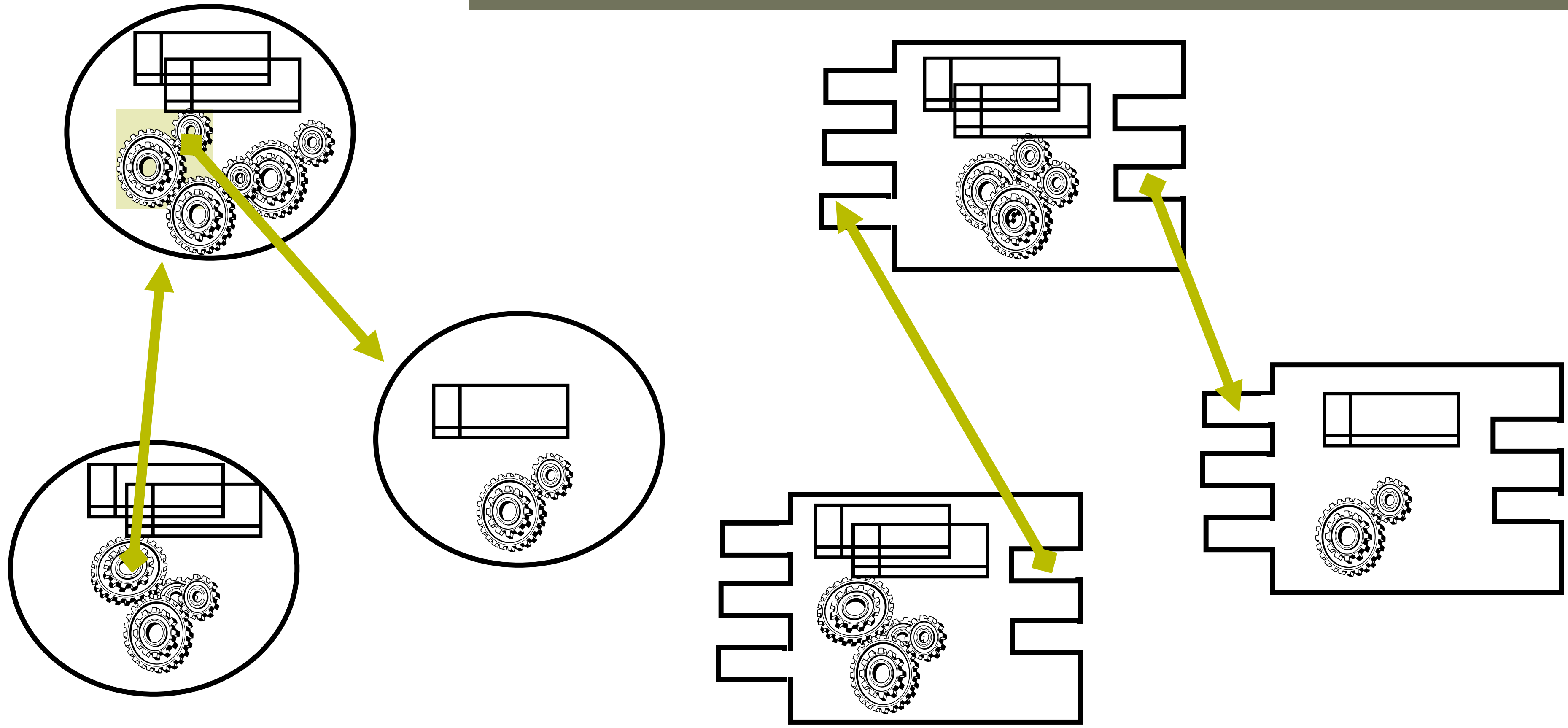


Component Based Development

- The ultimate re-use scenario: application development as black-box assembly of off-the-shelf components (COTS)
- The use of components is a law of nature in any maturing engineering discipline, the cost of component building can be spread out over multiple applications
- The concept of component software brings the middle path between custom-made and standard software: each component is a standardised product with all the advantages this brings while the process of component assembly allows for significant customisations
- Software components are binary units of independent production, acquisition, and deployment
- Software differs from other engineering discipline as it is the blue print that is delivered rather than the realisations of it

Components:

coarse grained, third party, black-box, explicit interfaces, QoS



Interfaces and Explicit Context Dependencies

- The interface of a component defines its access points, clients (usually other components) access services provided by a component through these access points
  - **Syntax:** specification of the *provided* interfaces using some standard Interface Description Language
  - **Semantics:** (formal) specification of the functionality of each service provided (e.g. pre- and post-conditions)
  - **Syncronisation:** (formal) specification of expected or imposed ordering, grouping and mutual exclusion of services provided
  - **Quality of Service:** specification of guaranteed response times, upperbounds for resource consumption (CPU-time, memory, etc.), failure rates, mean time between failure, etc.
- The component must specify what the deployment environment must provide for the component to be able to function properly
  - Required interfaces of other components.
  - Since there are multiple component world emerging, components must also mention the world they are prepared for (I.e. platform, implementation language, component model, component and library versions etc.)

Component "Weight"

In CBSD components have to be loosely coupled

Fat Components

- The component is self-contained and can function under weak environmental guarantees
- The context dependencies are reduced making the component more robust over time
- But a component with everything bundled in is not a component anymore

Lean Components

- Other components are (re)-used to achieve the component's services
- The context dependencies increase making the component more vulnerable in case of context evolution
- Re-use is maximized, use is compromised

Scale and Granularity

- A component's size may vary from a single class or function to a complete subsystem
- Most of the aspects relevant to granularity seem to demand fairly coarse grained partitionings

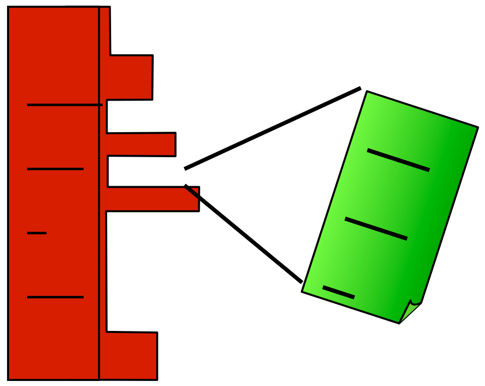
Components as units of
abstraction, accounting, compilation,
delivery, dispute, fault containment, instantiation,
loading, maintenance, system management

Component Composition or Wiring (1)

- Most class libraries and frameworks are not components in the strict sense, they are delivered in source form and implementation inheritance is the common re-use mechanism which is typically white-box reuse

Component Composition or Wiring (1)

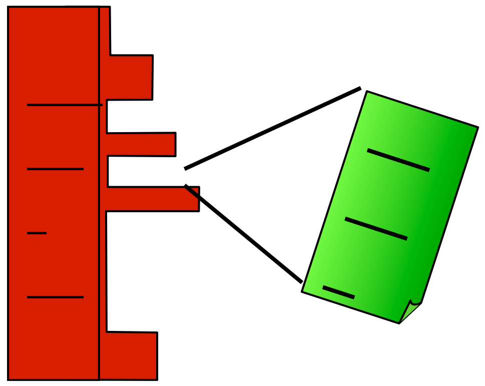
- Most class libraries and frameworks are not components in the strict sense, they are delivered in source form and implementation inheritance is the common re-use mechanism which is typically white-box reuse



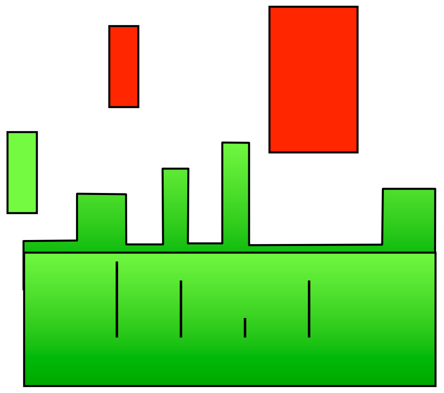
Library approach

Component Composition or Wiring (1)

- Most class libraries and frameworks are not components in the strict sense, they are delivered in source form and implementation inheritance is the common re-use mechanism which is typically white-box reuse



Library approach

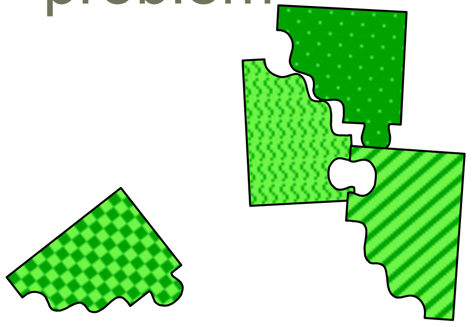


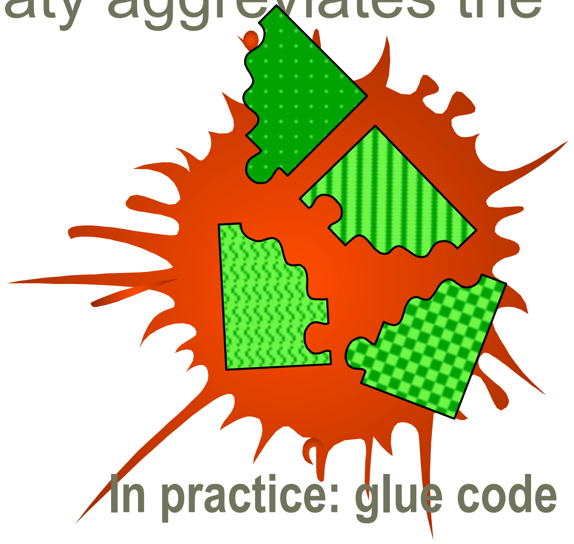
Framework approach

Vrije Universiteit Brussel

Component Composition or Wiring (2)

- Software components are third party configurable: blackbox reuse with plug-and-play composition is aimed for
- In practice, lots of glue code needs to be written to make components work together
- Distribution and heterogeneaty aggreviates the problem

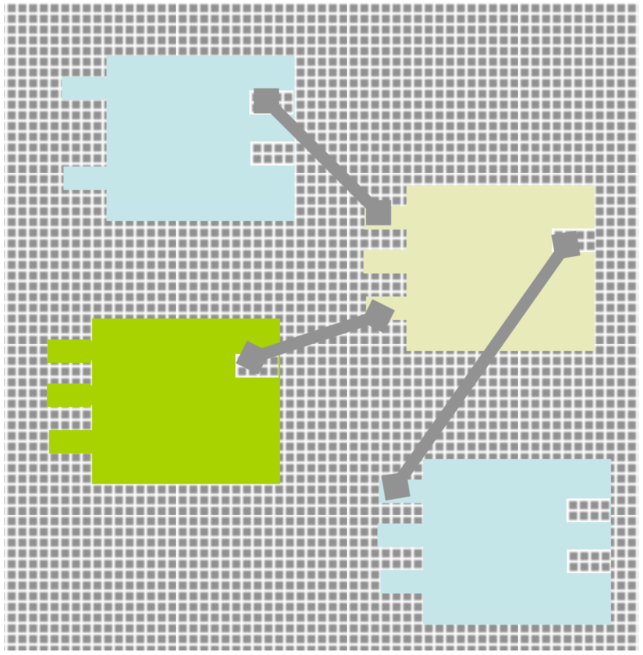

Utopic approach: plug and play


In practice: glue code

Vrije Universiteit Brussel

Middleware = Component Infrastructure Technology

- Utopic scenario:
  - Select&Wire
  - Go
- Real world:
  - Interface mismatches
  - Heterogeneous, distributed platforms



- Middleware
  - Introduces some component-connector model
  - Supports interoperation of components over heterogeneous systems
  - Includes reusable services (persistency, security, transaction management, etc.)
  - Offers deployment and execution environment
    - Find out what components are currently connected
    - make references to components via some naming scheme
    - guarantee once-only delivery of messages between components

Technical Solutions

Middleware

- CORBA
- COM/DCOM
- J2EE
- Web Services

OMG

Microsoft COM TECHNOLOGIES

Sun microsystems

W3C® WORLD WIDE WEB consortium

Component models

- JAVA BEANS
- EJB
- Active-X

activex

Vrije Universiteit Brussel

What’s New?

- Distributing Computing = Teamwork among Computers
- To make distributed *programs* we need Remote Procedure Calls (RPC)
- The first generation of RPC made the network transparent for function invocations
- Remote Method Call (RMI) is the OO-variant
- Middlewares do that AND support transfer of whole objects across network connections, between different platforms, running programs in different languages, provide extra services, etc.

Case:

CORBA

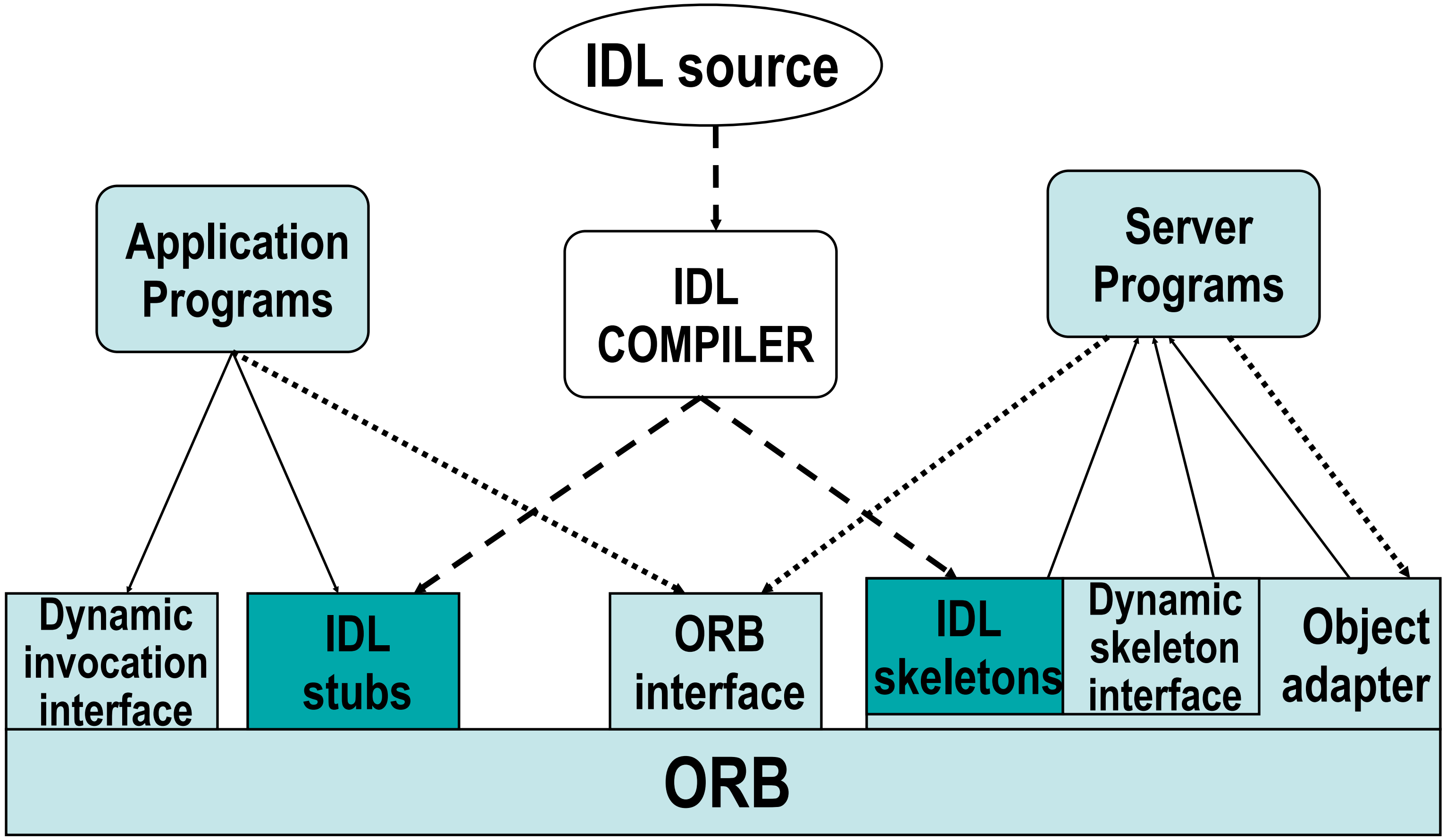
OMG's Corba

- OMG is a large consortium in the computer industry that operates as a non profit organisation and aims at the standardisation of "whatever it takes" to achieve interoperability of object-oriented systems implemented in different languages on different platforms
- The outcome is the Common Object Request Broker Architecture which is an open standard
- Corba has essentially three parts: a set of invocation interfaces, the Object Request Broker (ORB), and a set of object adapters

OMG's IDL

- For invocation interfaces and object adapters to work all interfaces must be described in a common language and all languages must have a mapping to this common language, OMG's IDL is this common language
- Once interfaces are written in IDL they are compiled and put in a repository which resides with the ORB. Program fragments that implement these interfaces are compiled and put in an implementation repository also with the ORB
- An ORB specific IDL compiler is used to generate stubs and skeletons (client and server side proxy's)

Object Request Broker Architecture



Case:

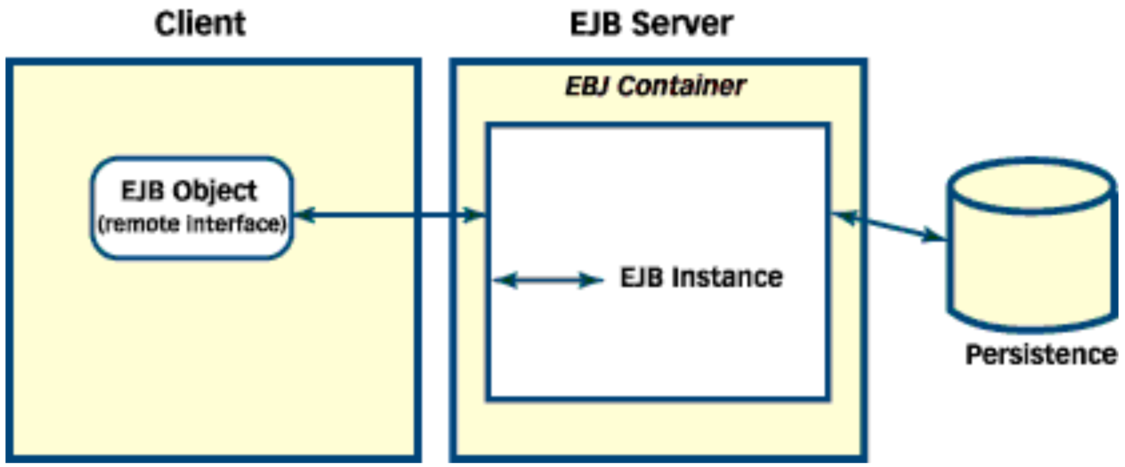
JEE

J2EE (now Java EE) middleware for the Java world

- Industry standard for developing portable, robust, scalable, multi-user, and secure server-side Java applications
- Builds on the **Enterprise Java Beans** component model
- EJB is designed to make application creation easy, I.e free programmers from details of managing transactions, thread, load balancing, etc.
- Allows to combine components from different vendors, to combine with non-Java applications and interoperates with Corba

EJB basics (1)

- **EJB component:** A Java class written by a developer, implements business logic, lives in a EJB container that runs on a EJB server
- **EJB container:** Resides on the server and provides services such as transaction and resource management, versioning, scalability, mobility, persistence…
- **EJB object and the remote interface:** An EJB object resides on the client and remotely executes the the EJB components's methods (proxy). *(The EJB object is created by code generation tools that come with the EJB container).*



EJB basics (2)

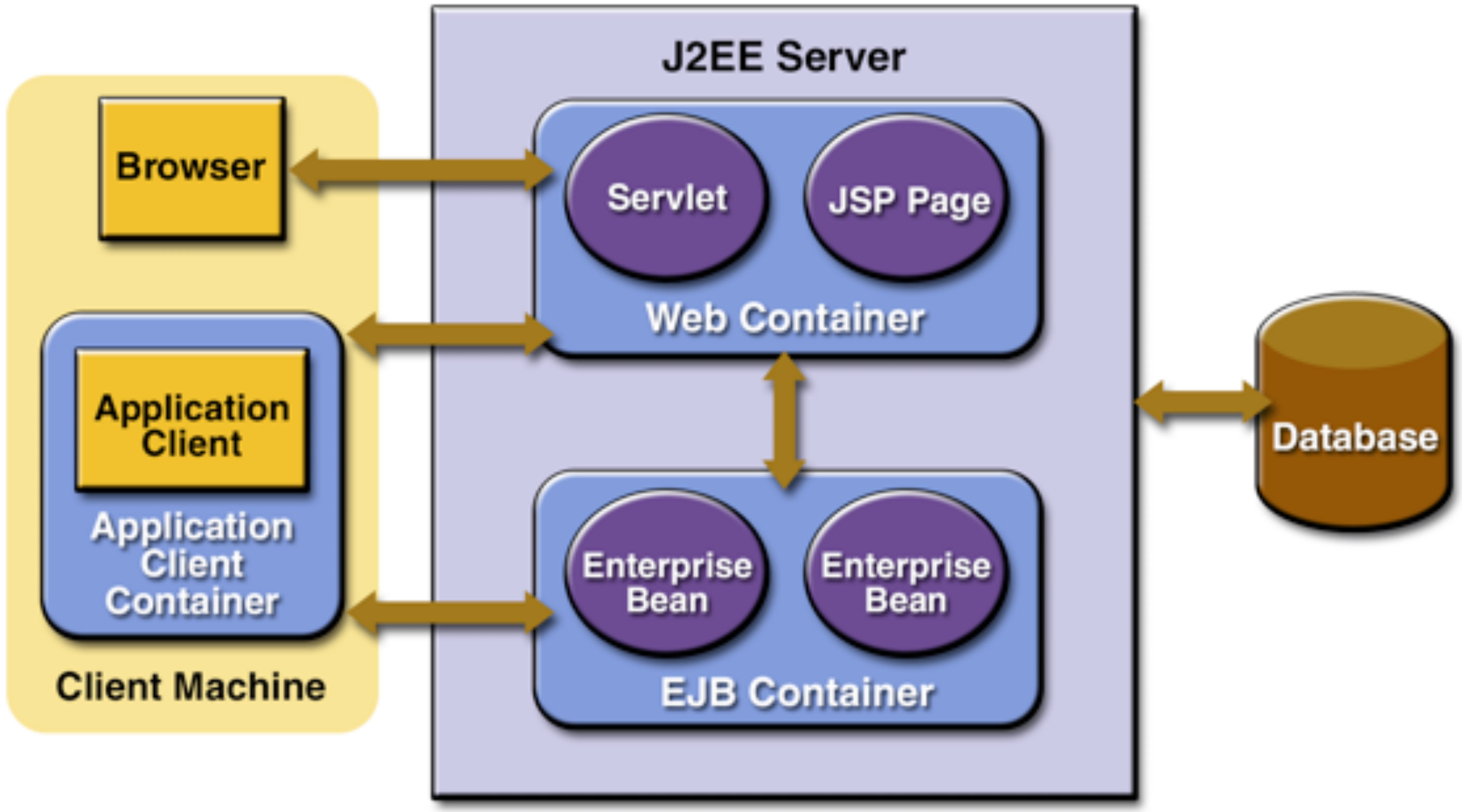
- **Two types of Enterprise JavaBeans**
  - **Session Beans:**
    - Associated with a single client
    - Typically not persistent, will not survive server crashes
  - **Entity Beans:**
    - Represent information persistently stored in a database
    - Associated with database transactions
- **The home interface**
  - Each EJB component has a home interface that defines methods for creating, destroying and (in case of entity beans) locating EJB instances
  - The EJB container is responsible for the life-cycle of server-side objects, e.g. a client request a container to create an instance of a particular EJB component and the container installs an instance and returns an EJB object to manipulate the instance
  - The Java Naming and Directory Interface (JNDI) is used by clients to locate the home interface for the class of beans it wants to use

Vrije Universiteit Brussel

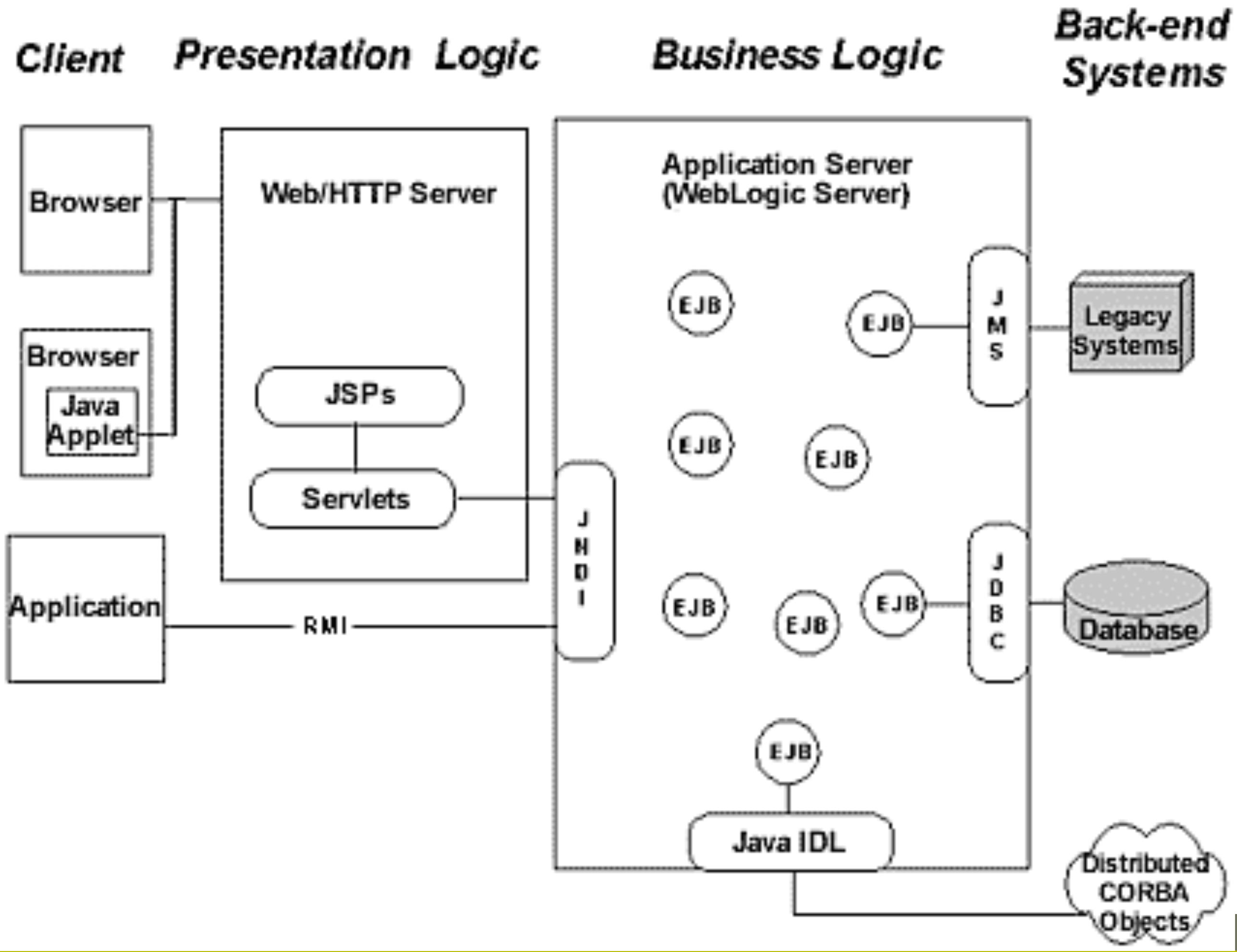
J2EE deployment descriptors

- Deployment descriptors describe the contents of deployment units and configure components and applications to their environment.
- In J2EE a deployment descriptor is a text based XML file that conforms to the deployment descriptor's XML schema as defined in the J2EE specification
- J2EE modules have deployment descriptors specific to the module type (EJB components, Web components, Client components, resource adaptors, libraries, …)
- J2EE applications have their own deployment descriptor format. The application programmer in charge of combining and packaging one or more modules into a J2EE application is responsible for providing the deployment descriptor associated with the developed application
- A deployment descriptor contains information about:
  - Components and modules that are used
  - Initialisations
  - Persistency type (in an Entity Bean deployment descriptor)
  - Security roles (in an application deployment descriptor)
  - . . .

J2EE global architecture



N-tier architectures with J2EE technology



Case:

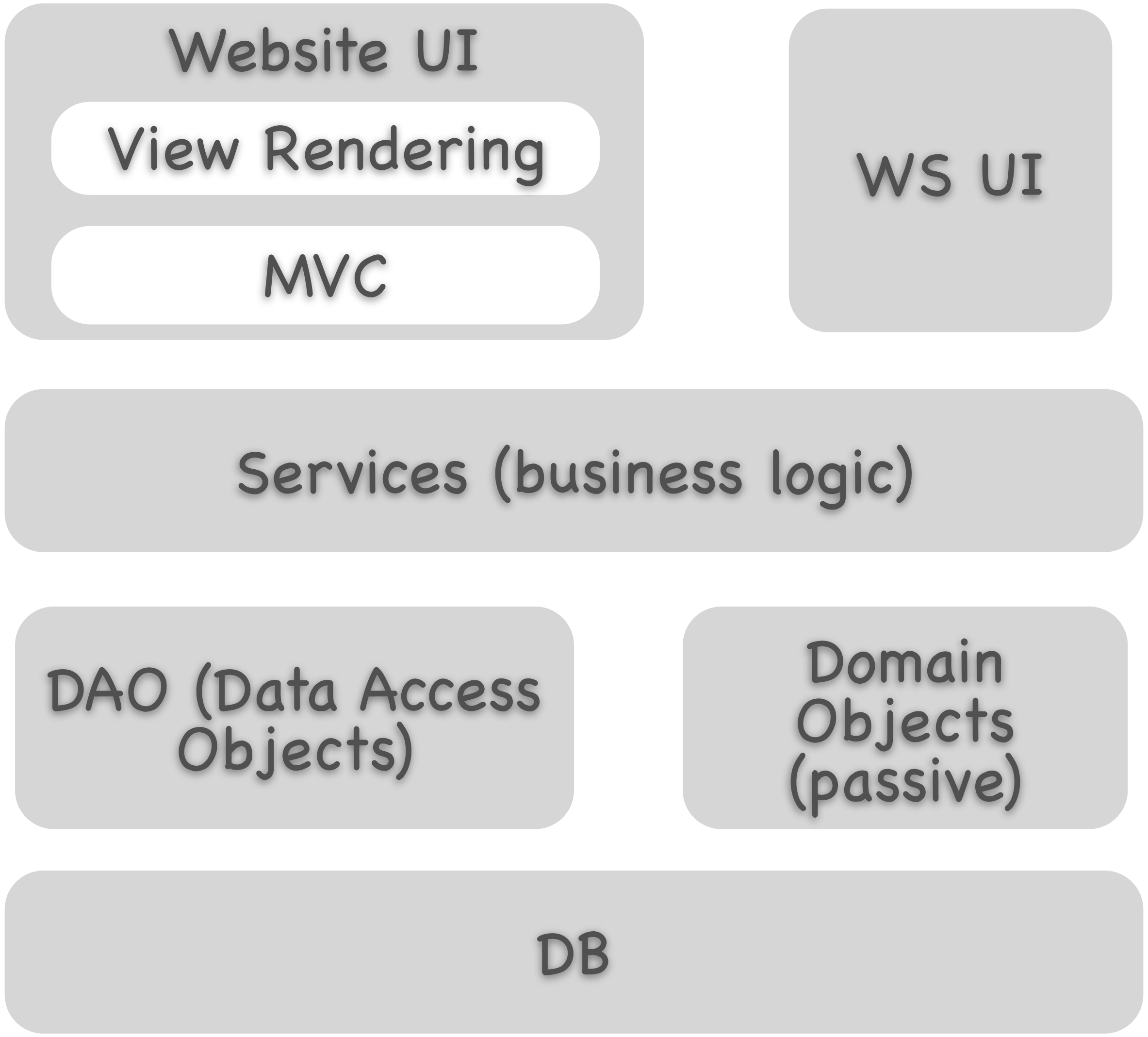
SPRING

Spring

- A Layered Java Application framework
- Plain POJO beans instead of EJB
- Dependency injection instead of lookup
- Convention over configuration
- Abstraction layers for external APIs
- Compatible with a large range of application servers
- http://www.springsource.org/

Vrije Universiteit Brussel

Typical Spring Architecture



POJO Bean??

- Plain Old Java Object
- But with a couple of naming conventions:
  - A setter for property *prop* is named *setProp*
  - A getter for property *prop* is named *getProp*
- Example:

```
public class MyComponent {
  private String name;

  public String getName() { //the getter
    return name; }

  public void setName(String name) { //the setter
    this.name=name; }
}
```

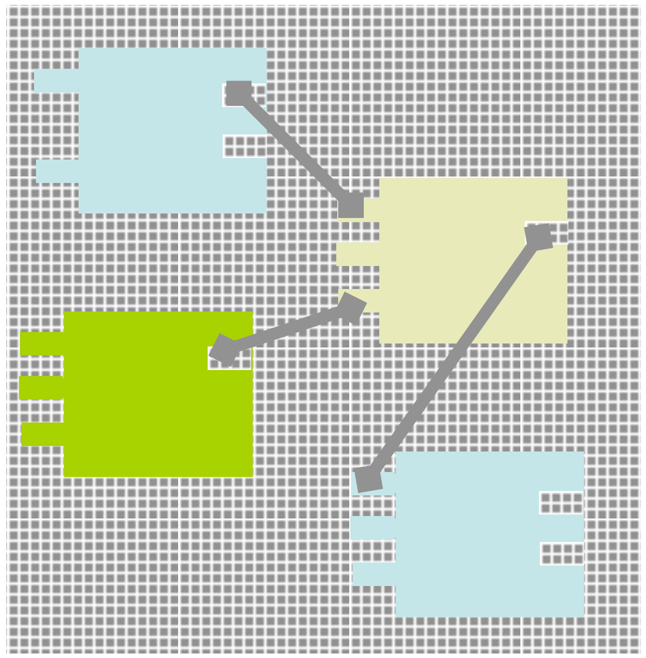
Dependency Injection?

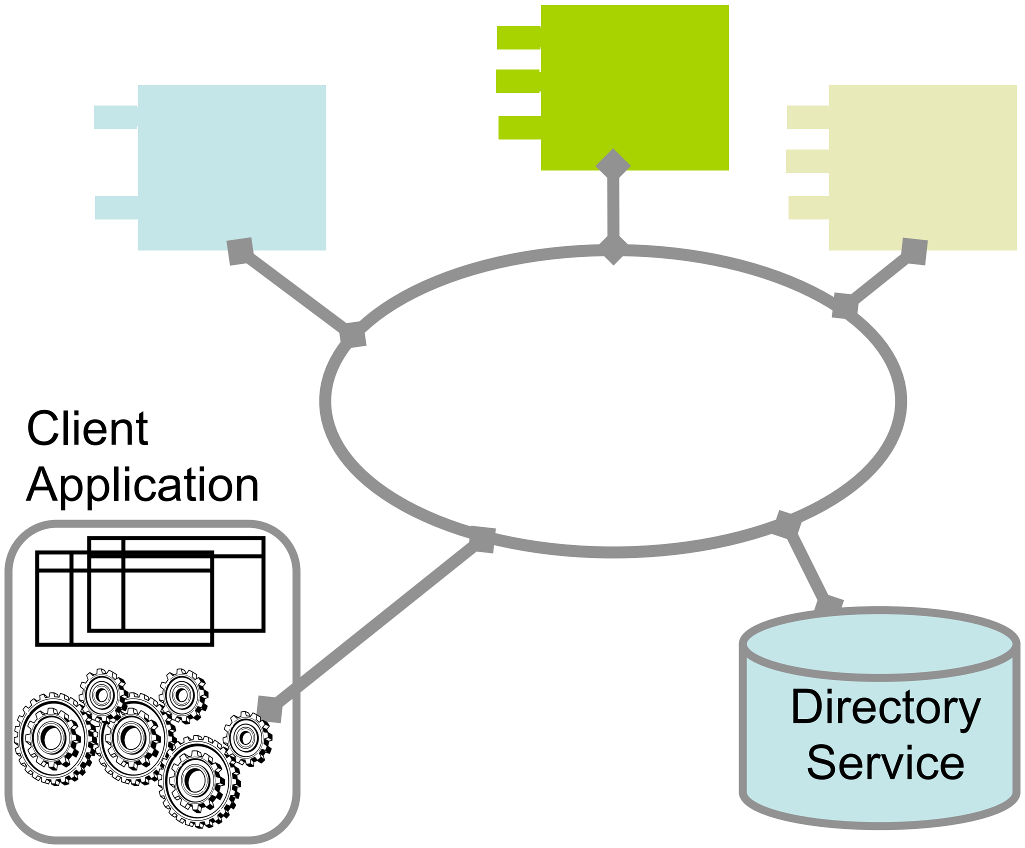
- References and properties are injected by the container
- Container follows the composition specified in a Spring Beans Configuration file (XML)

Service Oriented Architecture

Services:

more coarse grained components, self-contained, more loosely-coupled

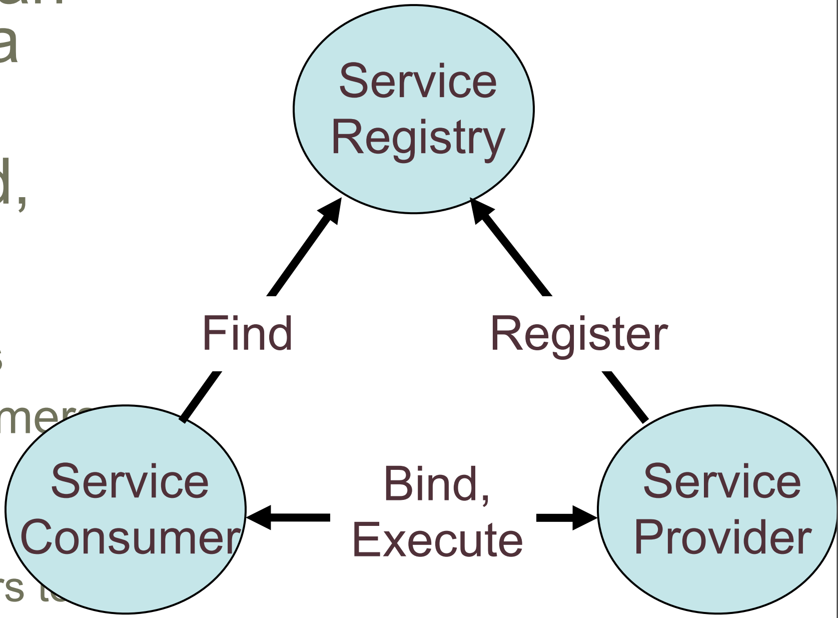




Vrije Universiteit Brussel

What is Service Oriented Architecture (SOA)?

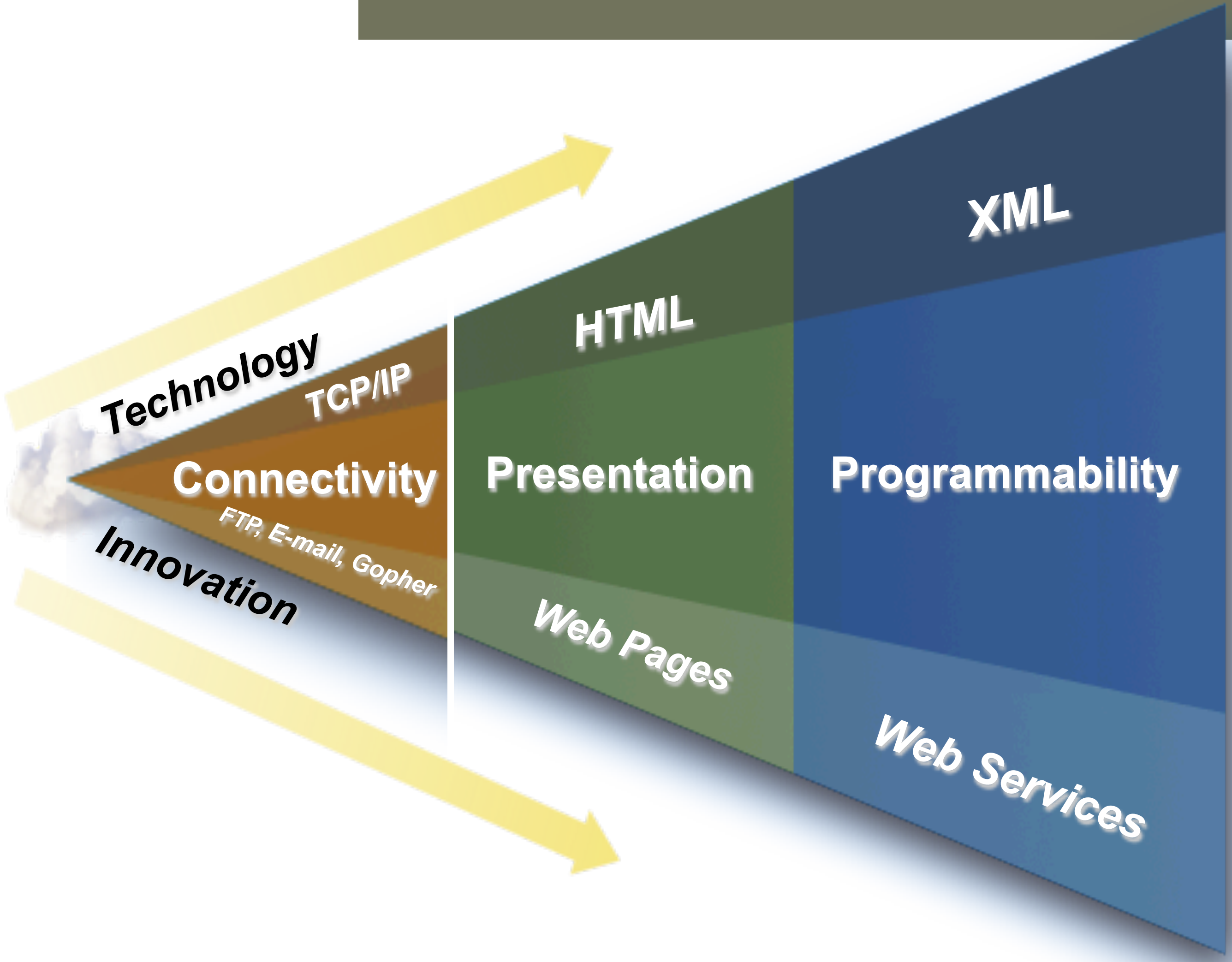
- An SOA application is a composition of services
- A “service” is the atomic unit of an SOA; one service encapsulate a business process
- Service use involves: Find, Bind, Execute
  - Service Providers provide stateless, location transparent business services
  - Service Registry allows service consumers to locate service providers that meet required criteria
  - Service Consumers use service providers to complete business processes
- Most well-known instance is Web Services



Why is SOA different?

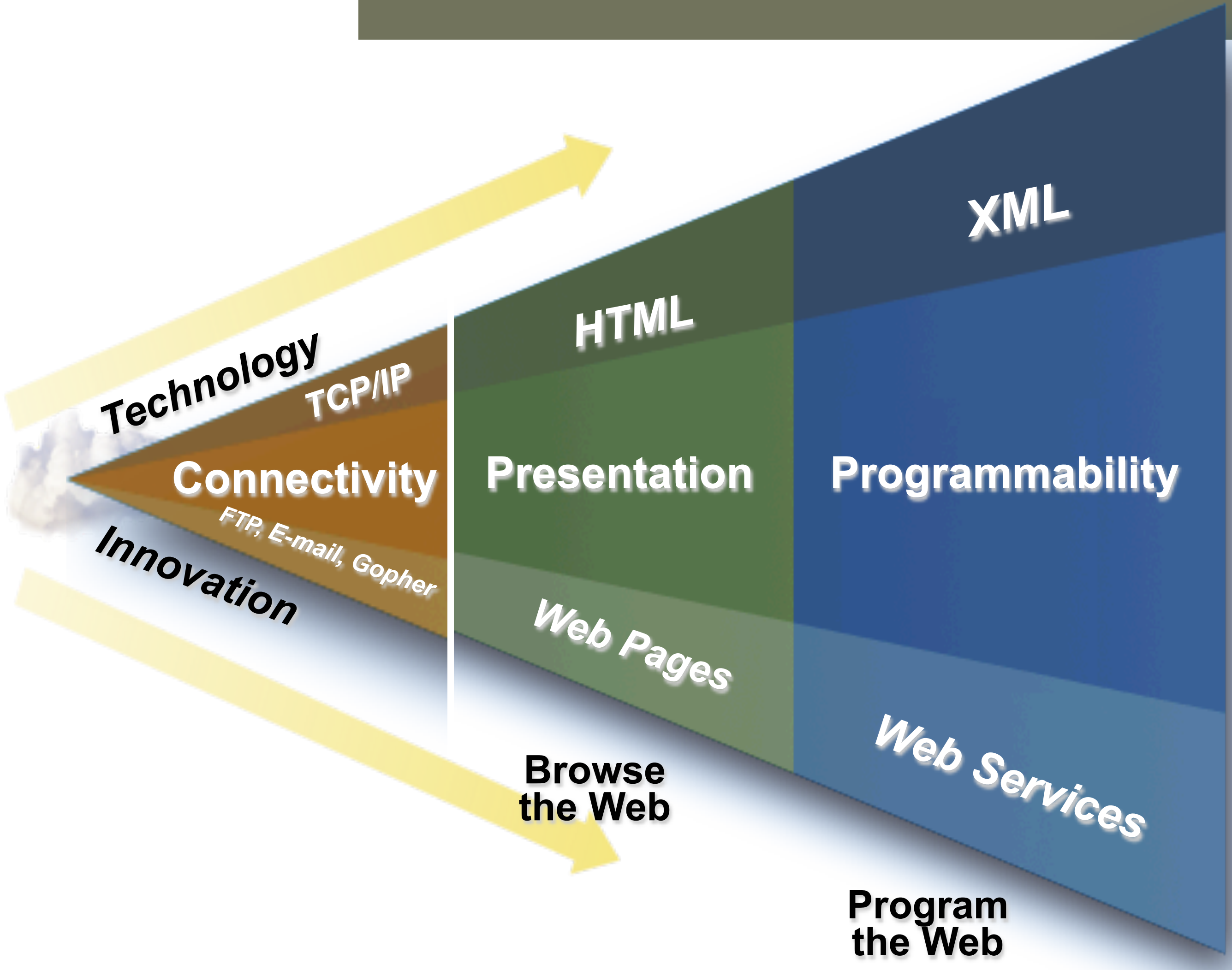
- SOA reflects the reality of ownership boundaries
  - CORBA, RMI, COM, DCOM, etc. all try to implement *transparent* distributed systems
  - Ownership is of the essence in SOA
- SOA is task oriented
  - Services are organized by function
    - Getting something done
- SOA is inspired by human organizations
  - It worked for us, it should work for machines

Web Services



Vrije Universiteit Brussel

Web Services

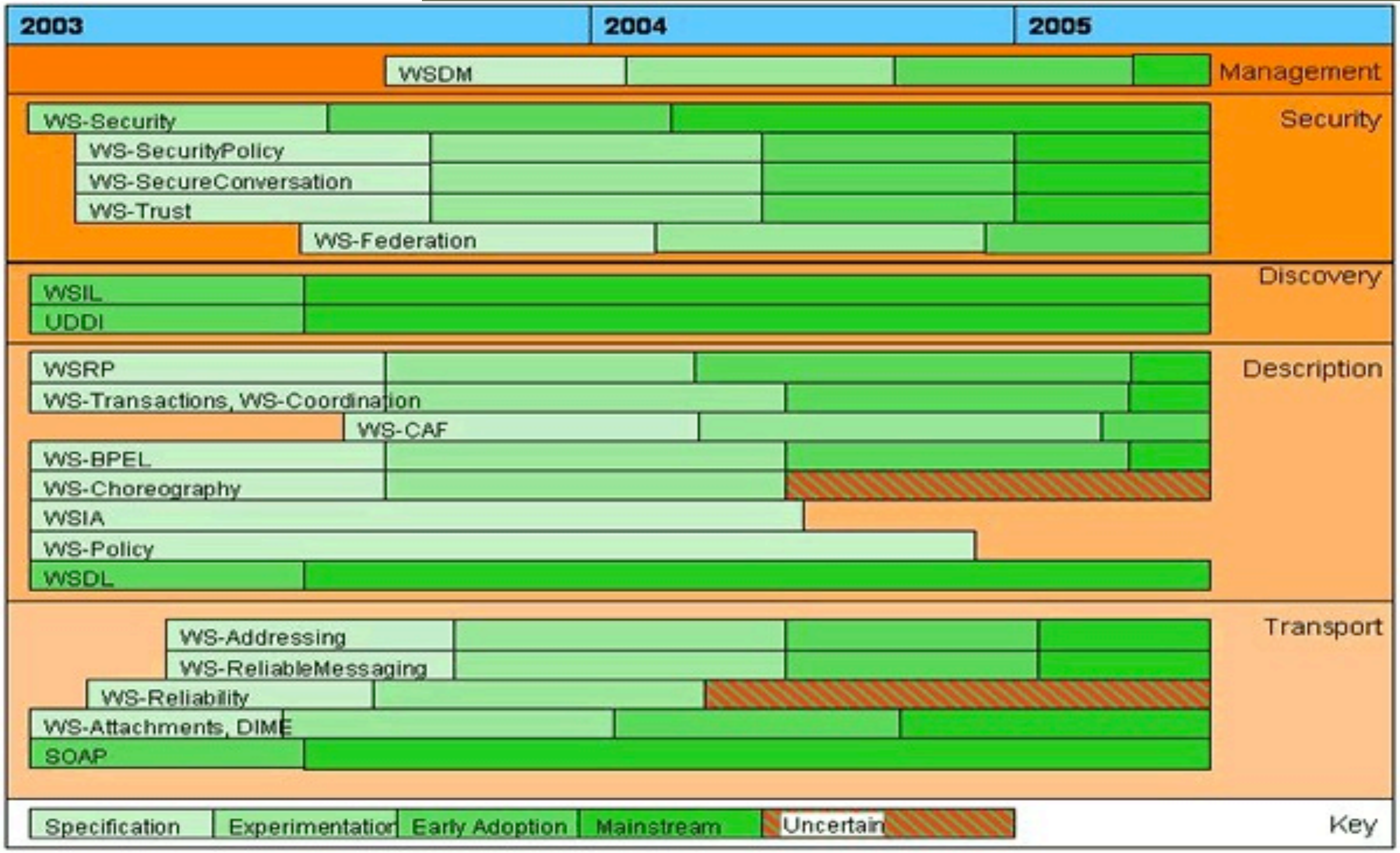


How Is It Done in Web Services?

- We need a protocol to transport data and function calls over the network (i.e. to support RPC)
  - SOAP (Simple Object Access Protocol) over HTTP
- We need to find out what function calls and parameters are expected by a given web service.
  - WSDL (Web Service Description Language)
- We need to find out which web services there are
  - UDDI (Universal Description, Discovery and Integration Service)
  - (Today often informally: go there and there to find the WSDL file …)

Vrije Universiteit Brussel

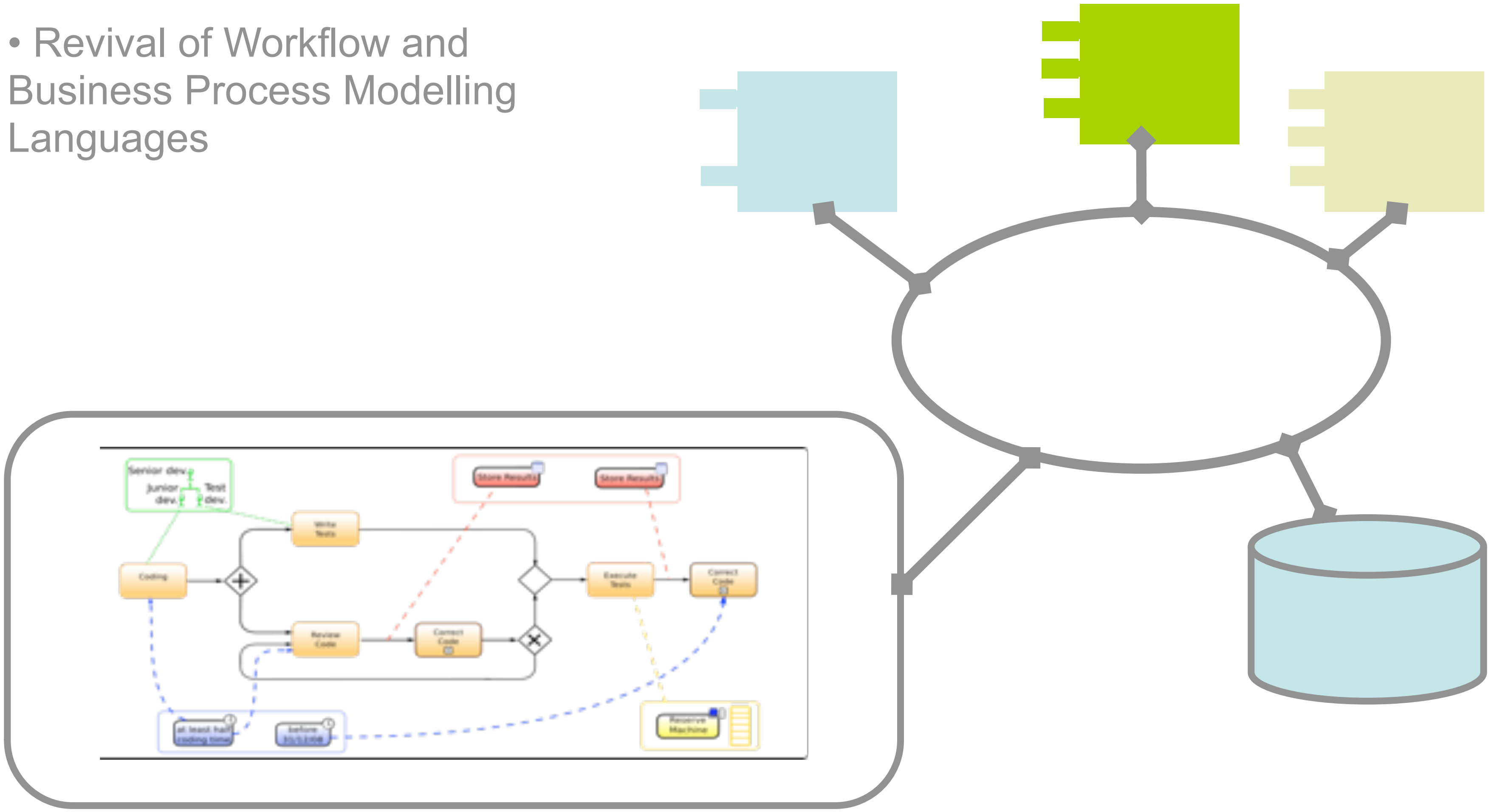
SOA/Web Services Related Standards



Source: http://roadmap.cbdiforum.com/reports/protocols/

Service Oriented Architecture: service composition & orchestration

- Revival of Workflow and Business Process Modelling Languages



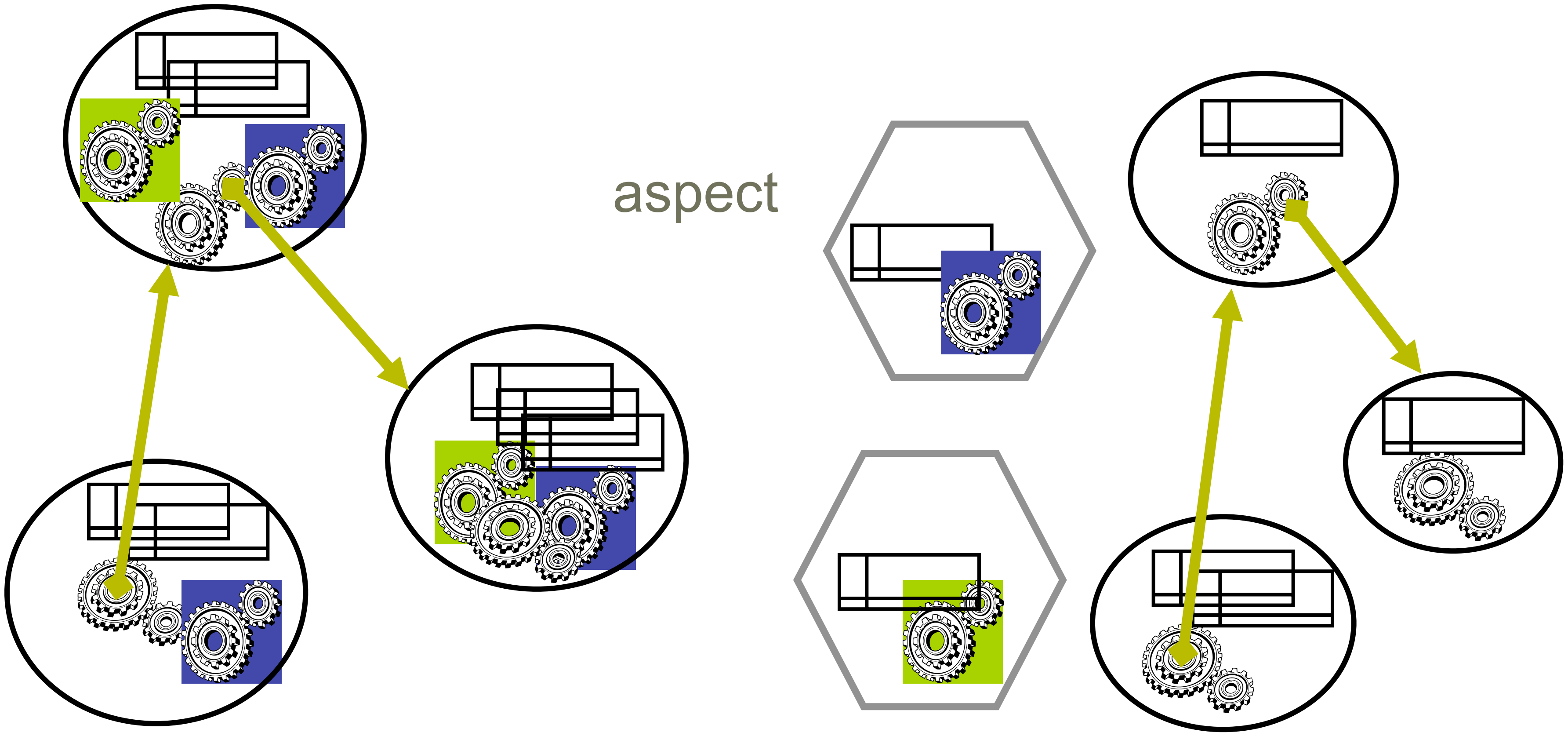
AOSD

AOSD

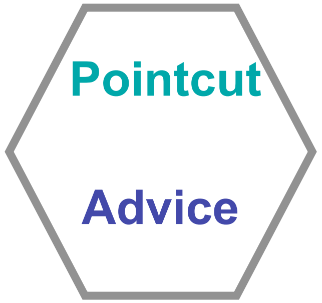
- crosscutting is inherent in complex systems

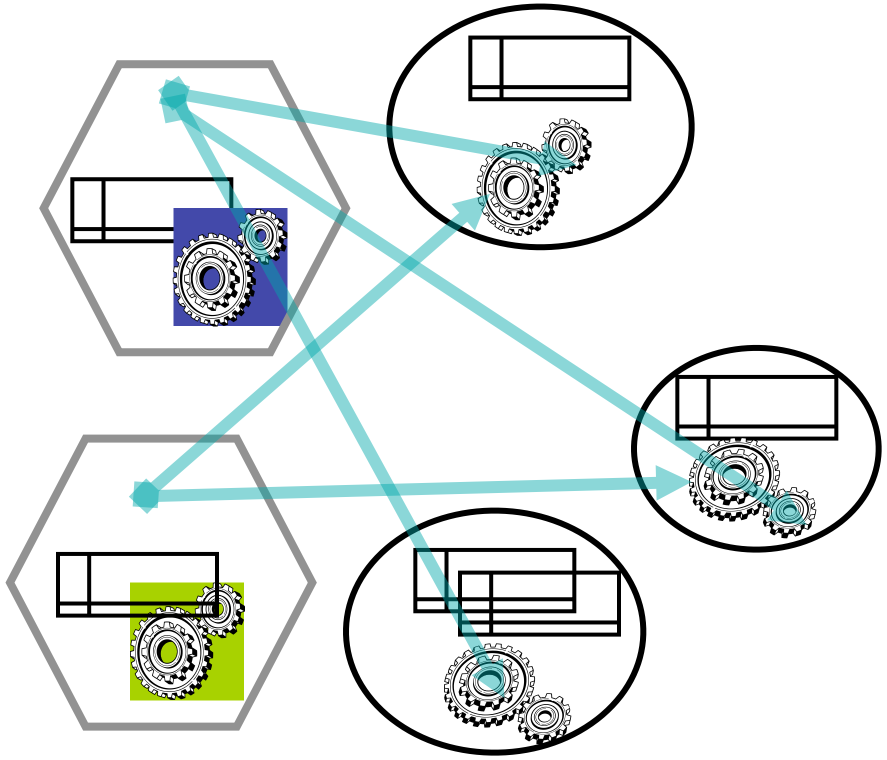
  “tyranny of the dominant decomposition”
- crosscutting concerns
  - have a clear purpose ***What***
  - have some regular interaction points ***Where/When***
- AOSD proposes to capture crosscutting concerns explicitly...
  - in a modular way
  - not only in programming languages but in all stages of software development
  - and with appropriate tool support

Aspect-Oriented Programming:
modularisation of crosscutting concerns



Aspect-Oriented Programming: inversion of control





- Pointcut describes a number of joinpoints, i.e. points of interest in the base program [***Where/When***]
- Advice is extra code to be executed (before-after-instead) a jointpoint is reached [***What***]

AO Programming

JAsCo, CaesarJ, AspectS, Object Teams, HyperJ, JBOSS AOP, Compose*, DemeterJ, AspectC++, ...

- Aspectual language features
  - Advice models
  - Join point models
  - Pointcut languages
  - Weaving: a technology for bringing aspects and base code together
- Development support
  - IDE's

Closing the circle

- Application server middleware supports separation of concerns in a (limited) way
- AOP for middleware: Application servers are a killer application for AOP. Implementing sophisticated, flexible, and extensible middleware would benefit from AOP facilities
- Middleware for AOP: AOP frameworks emerge that build AOP facilities in or upon application server middleware

Spring AOP

Spring and AOP

- Spring explicitly supports AspectJ AOP
- Aspects can be configured like normal Spring components (dependency injection)
- Supported syntax:
  - XML-based definition
  - AspectJ annotation-based development style
  - AspectJ language
  - Domain Specific Languages for e.g. Transaction Management
- Aspect library

Spring AOP Weavers

- AspectJ weaver or built-in Spring weaver
- Built-in Spring weaver:
  - No external tools
  - Weaving happens automagically
  - Proxy-based:
    - only weaving on configured beans
    - as such domain classes are typically excluded from weaving
  - Only supports execution pointcuts
    - No call, field set, field get etc...

Spring/AOP
Syntax & Weavers

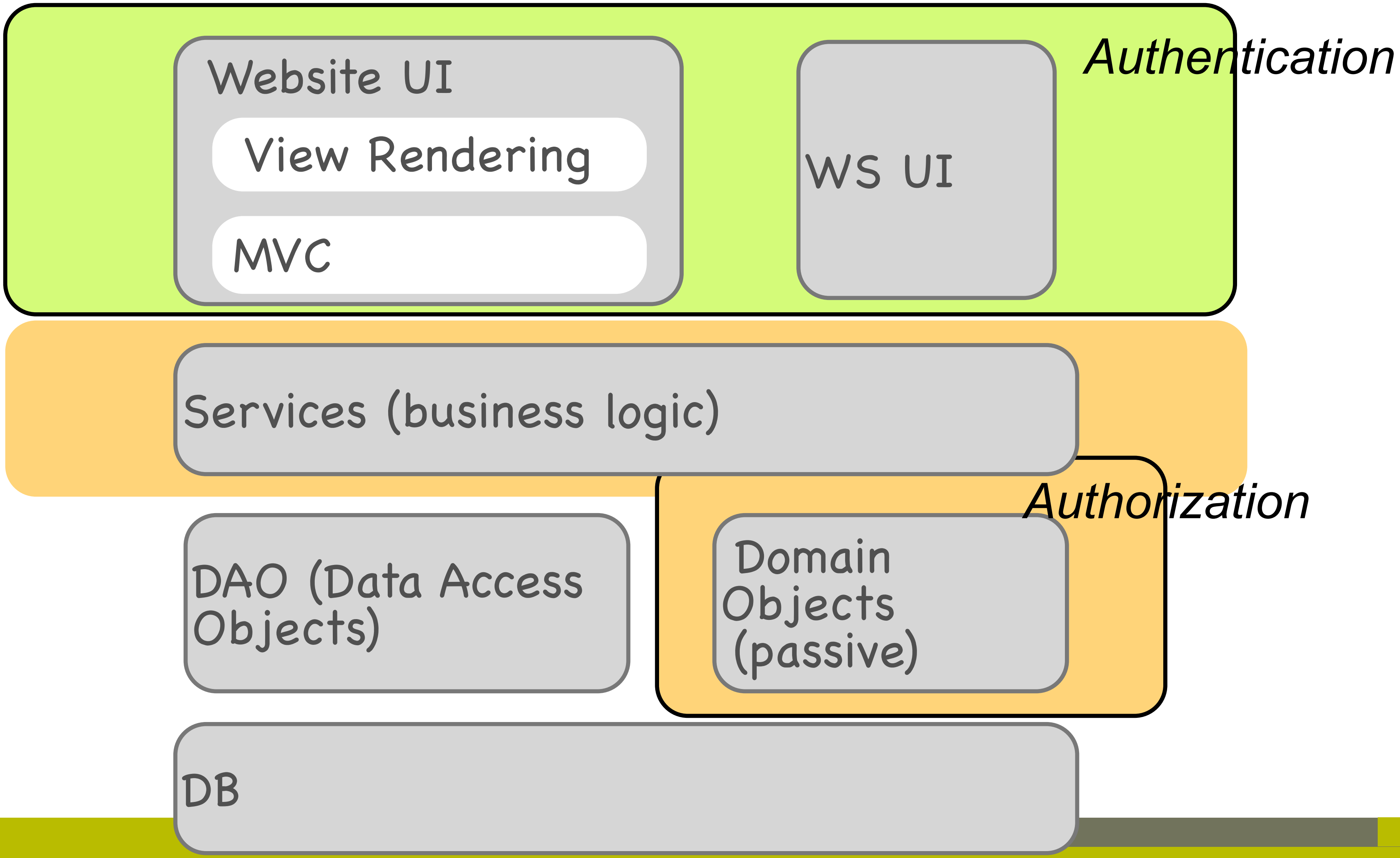
	AspectJ Language	AspectJ Annotation Style	XML Definition	DSL
Spring Weaver	No	Yes	Yes	Yes
AspectJ Weaver	Yes	Yes	No	No

Security

- Several facets:
  - Authentication: is the user who he says he is?
  - Authorization: is the user allowed to do a certain operation?
  - Confidentiality: make sure this data is not readable by non-authorized users

Vrije Universiteit Brussel

Security



Vrije Universiteit Brussel

Authentication Aspect

- For each controller invocation, check whether user has authenticated

```
around(XController c) : controllerInvocation(c) {
    if(isAllowedViewWithoutLogin(c.getViewName()))
        return proceed();

    else if(getCurrentAuthenticatedUser()==null)
        return getLoginView();
    else return proceed();

}
```

Authorization Aspect

- For each domain object invocation, check whether the current user has the correct credentials.

```
around(DomainObject o) : domainObjectInvocation(o) {
    if(hasAccess(getCurrentUser(),o))
        return proceed();
    else throw new SecurityException(....);
}
```