Software design and software architecture

Part 2

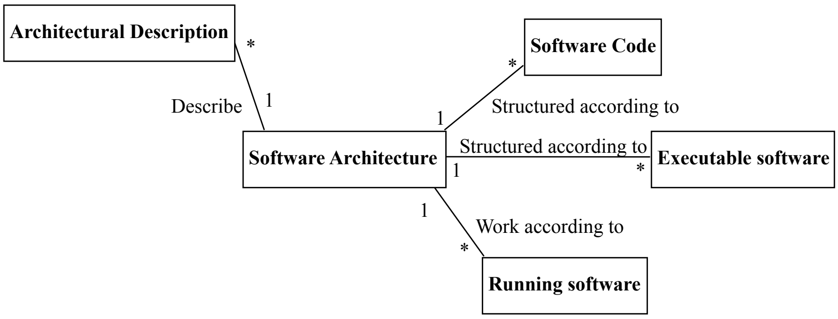
The focus

Here we focus on the software architecture at a high level of abstraction

- The meaning of architecture, architectural description, styles, component models

The architecture of a software system defines the system in terms of computational components and interactions among those components. (Garlan&Shaw1996)

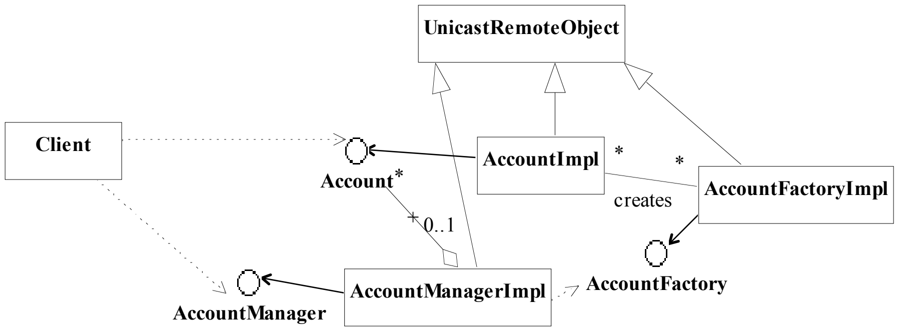
The software architecture as a central element of software



Components and interactions

- Can be defined at different levels of abstractions
- Let us consider a simple example taken from a banking domain

The level of the code: classes and interfaces (modules)



UnicastRemoteObject

Client

Account

*

*

0..1

AccountImpl

*

*

creates

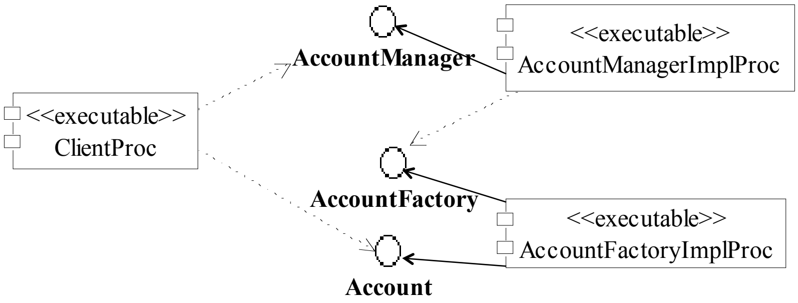
AccountFactoryImpl

AccountFactory

AccountManager

AccountManagerImpl

Executables (*deployment units*)



AccountManager

<<executable>>
AccountManagerImplProc

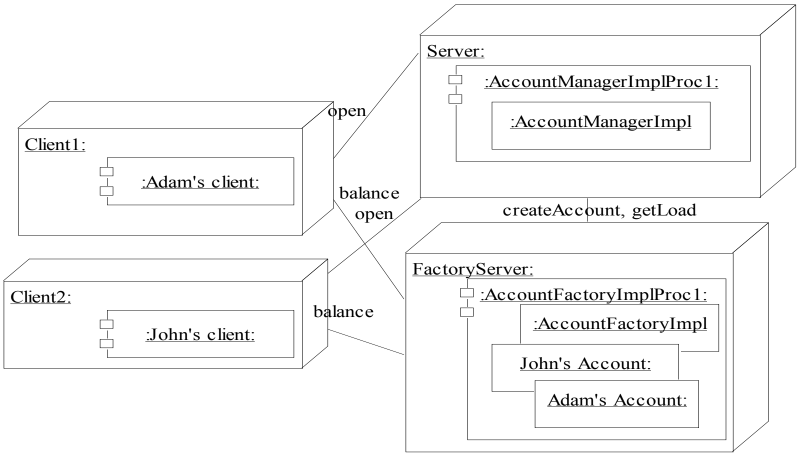
<<executable>>
ClientProc

AccountFactory

<<executable>>
AccountFactoryImplProc

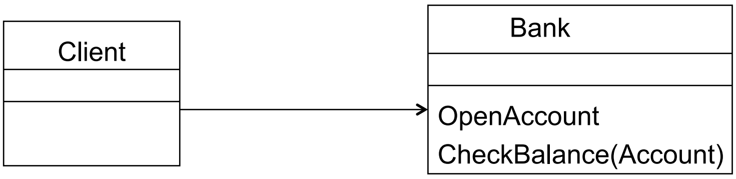
Account

Processes and threads (*runtime units*)

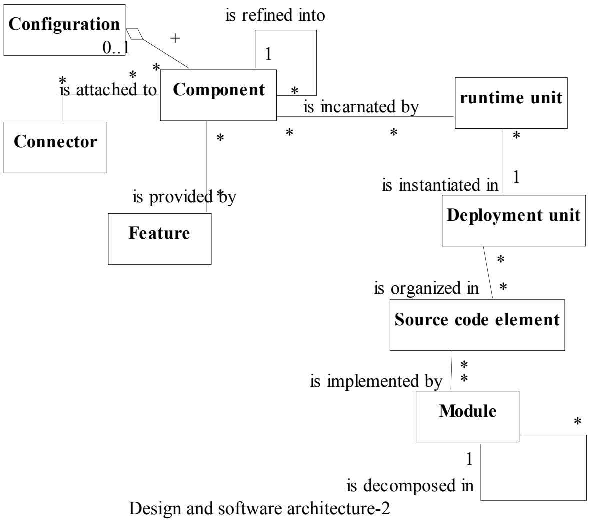


High-level functional view

- A Bank Server receives from clients two requests
  - Open account
    - returns an account identifier; if the account does not exist, the system creates it
  - Balance
    - checks the balance of a previously opened account
- Other details may be ignored at this level
  - The factory
  - The distinction between services offered by the manager and those offered by the account



Conceptual model



Configuration

Component

runtime unit

Connector

Feature

Deployment unit

Source code element

Module

is refined into

is attached to

is incarnated by

is provided by

is instantiated in

is organized in

is implemented by

is decomposed in

0..1

+

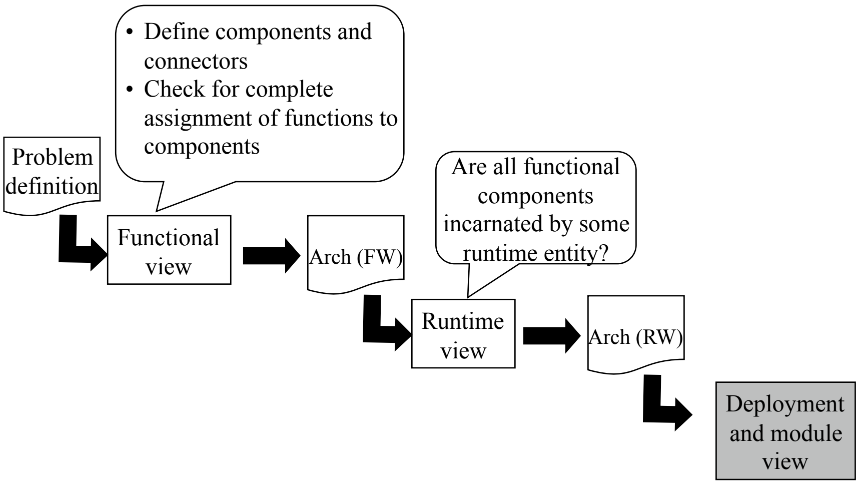
1

*

Views and architectural descriptions

- The most common views
  - *Functional view* defines the allocation of functionality to different components
    - A number of logical components are identified. Interaction between them is shown
  - *Runtime view* defines runtime units (the components available at runtime) and shows how they collaborate
  - *Deployment view* defines the main deployment units and the guidelines for their installation
  - *Module view* provides a logical decomposition of code in different *modules* (this is design in the small)

Architectural design: a top down process



Problem definition

Functional view

- Define components and connectors
- Check for complete assignment of functions to components

Arch (FW)

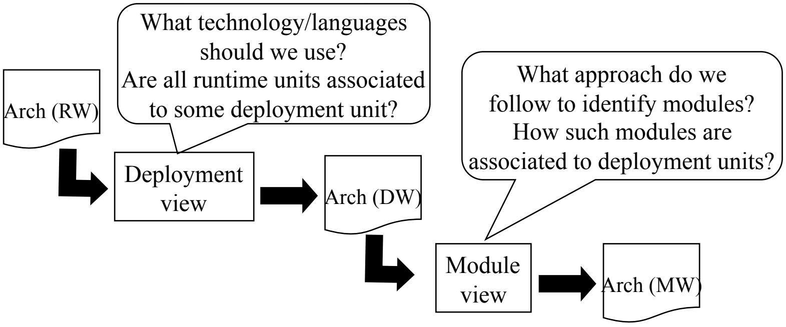
Runtime view

Are all functional components incarnated by some runtime entity?

Arch (RW)

Deployment and module view

Architectural design: a top down process (cont)



Arch (RW)

Deployment view

What technology/languages should we use?
Are all runtime units associated to some deployment unit?

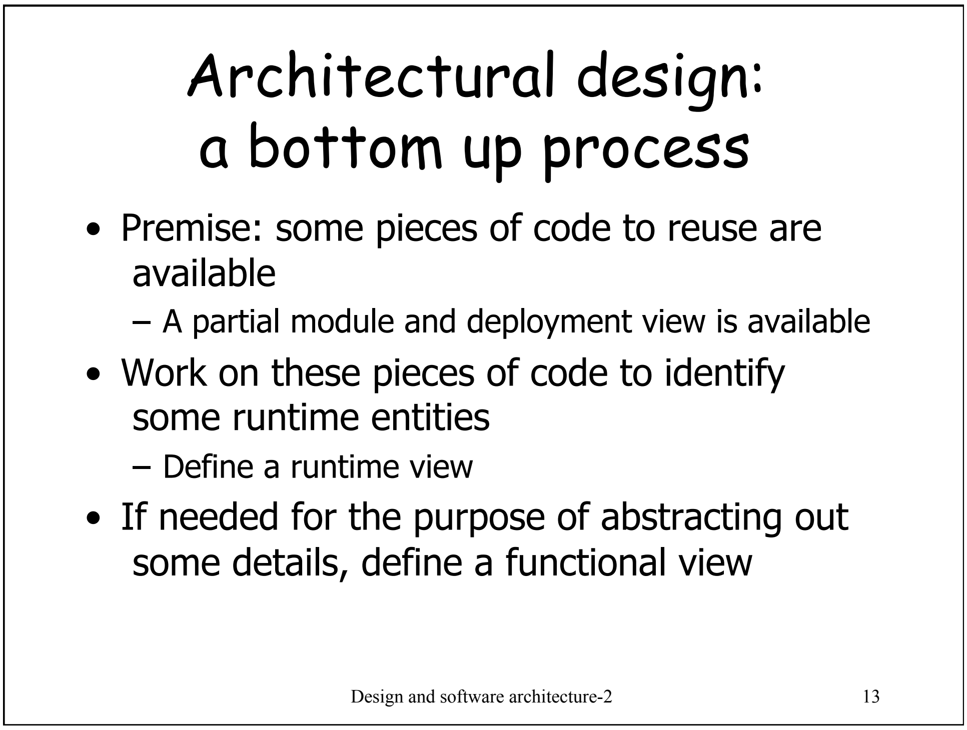
Arch (DW)

Module view

What approach do we follow to identify modules?
How such modules are associated to deployment units?

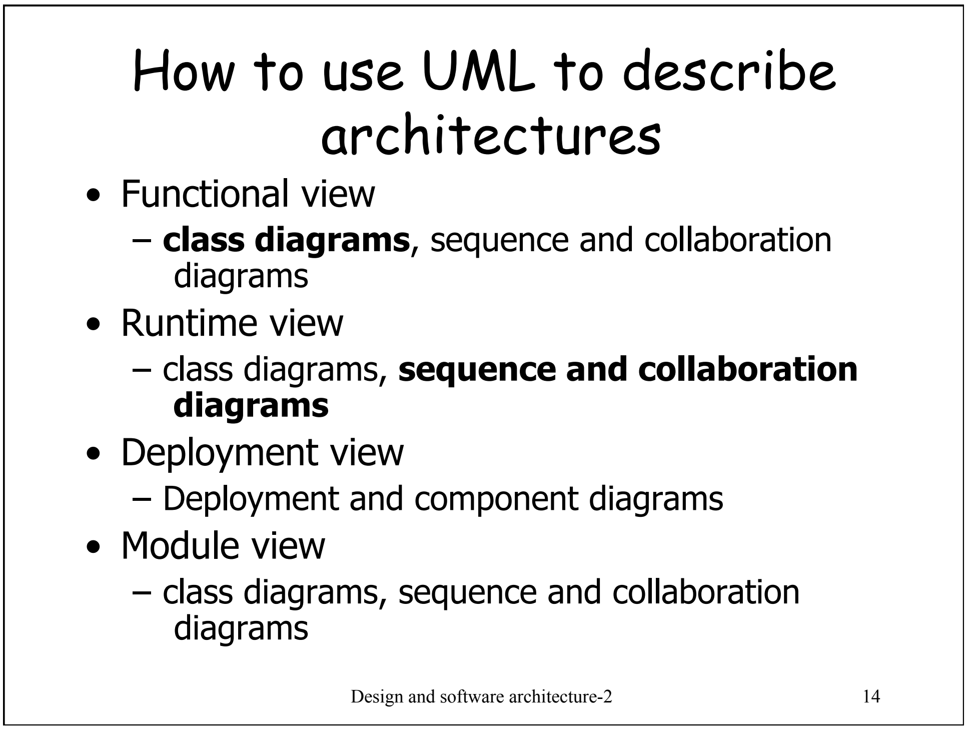
Arch (MW)

- NB: module view and, partially, deployment view belong to the detailed design phase



Architectural design: a bottom up process

- Premise: some pieces of code to reuse are available
  - A partial module and deployment view is available
- Work on these pieces of code to identify some runtime entities
  - Define a runtime view
- If needed for the purpose of abstracting out some details, define a functional view



How to use UML to describe architectures

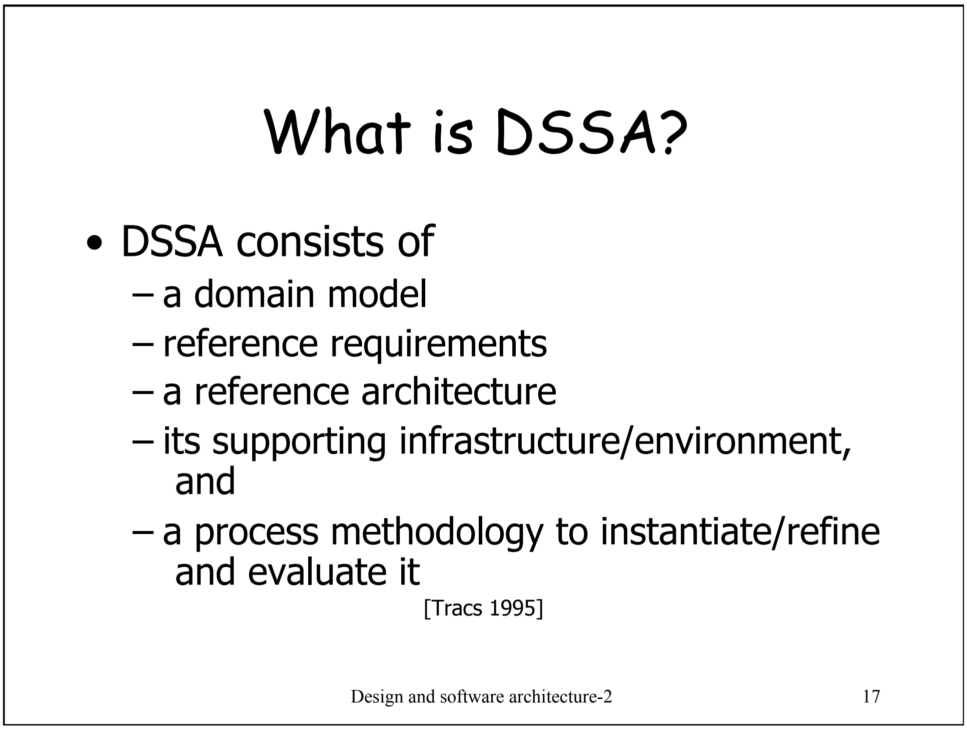
- Functional view
  - **class diagrams**, sequence and collaboration diagrams
- Runtime view
  - class diagrams, **sequence and collaboration diagrams**
- Deployment view
  - Deployment and component diagrams
- Module view
  - class diagrams, sequence and collaboration diagrams

Tools supporting architectural designers

- High level design
  - Reference architectures or Domain Specific Software Architectures (DSSAs)
  - Architectural styles
- Middleware and their component models
- Detailed design
  - Design patterns

DSSAs

- Architectures specifically developed for some domain
  - exploit commonality between systems providing similar functionality
  - minimize effort and risks
  - maximize consistency and coherency of the resulting system
  - apply reuse, obtain economies of scale
- Developed for well understood, critical domains, e.g.
  - avionics (ADAGE)
  - telecommunication (TINA)



What is DSSA?

- DSSA consists of
  - a domain model
  - reference requirements
  - a reference architecture
  - its supporting infrastructure/environment, and
  - a process methodology to instantiate/refine and evaluate it

[Tracs 1995]

Architectural Styles

Design styles

- Shared understanding of common design forms is typical of mature engineering fields
- Shared vocabulary of design idioms is codified in engineering handbooks
- Software is going in this direction
  - but there is less maturity

Style

Components are such things as clients and servers, databases, filters, and layers in a hierarchical system. Interactions among components can be simple and familiar, such as procedure call and shared variable access. But they can be complex and semantically rich, such as client-server protocols, asynchronous event multicast, and piped streams. (Garlan&Shaw 1996)

Components and connectors

- Components
  - clients
  - servers
  - filters
  - layers
  - databases
  - ...
- Connectors
  - procedure call
  - event broadcast
  - pipes
  - ...

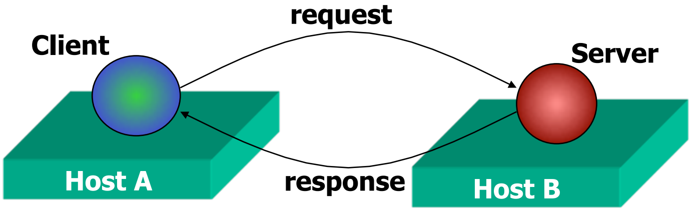
Client/Server

- The best known and most used architectural style for distributed applications

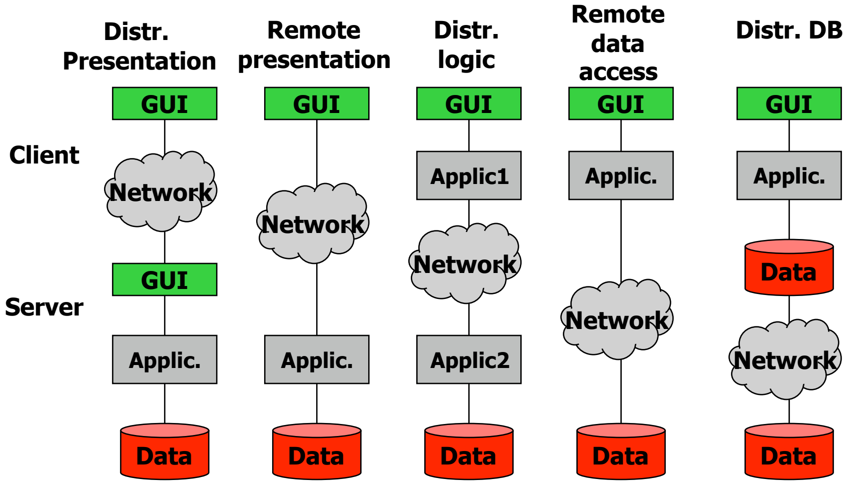
Client/Server

- Client & server are different processes with well-defined interface
  - Accessible only through interface
  - Client & server may be defined by a set of hw/sw modules
- Client & server play different roles
  - Server invoked to provide one of a set of services
  - Client uses them and initiates communication
  - Comunication through messages or through remote invocations

C/S: a general scheme



Two-tier architecture



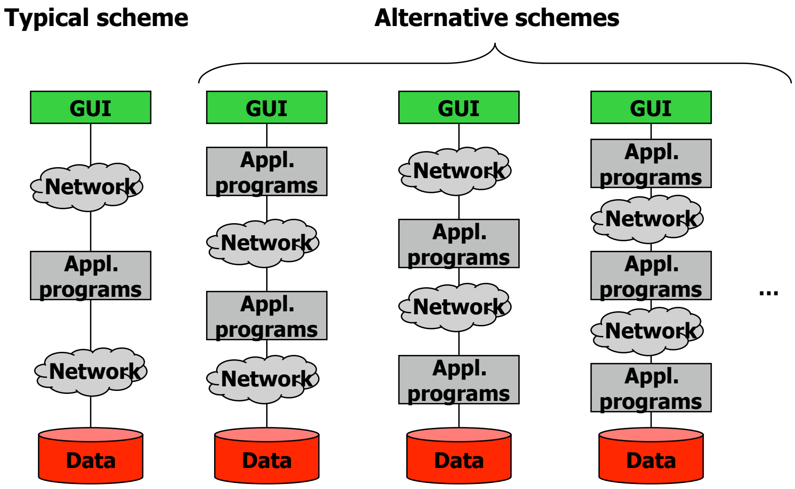
Two-tier architecture (1)

- Distributed presentation
  - "intelligence" in server, client only cares about presentation
    - es: pure HTML form cannot do input data validation
- Remote presentation
  - client in charge of all presentation issues
- Distributed logic
  - application logic partly decentralized in client

Two-tier architecture (2)

- Remote data access
  - client in charge of both presentation and logic
    - server used only to access data, typically through SQL interface
- Distributed DB
  - data management partly done by client and partly by server

Three-tier architecture

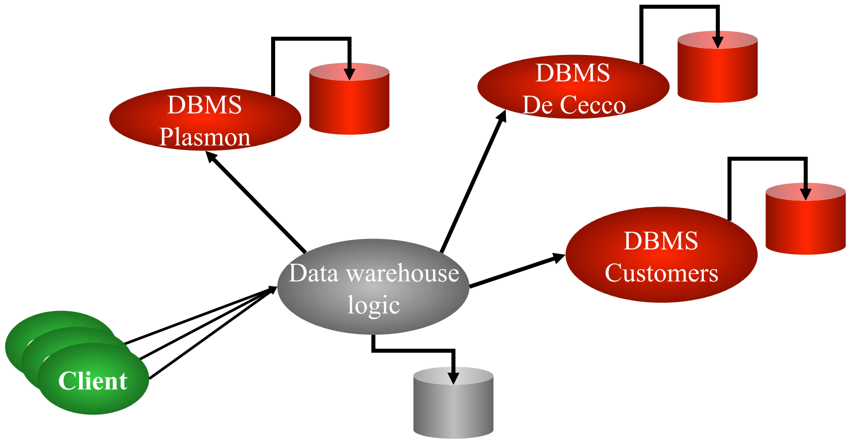


Three-tier architecture

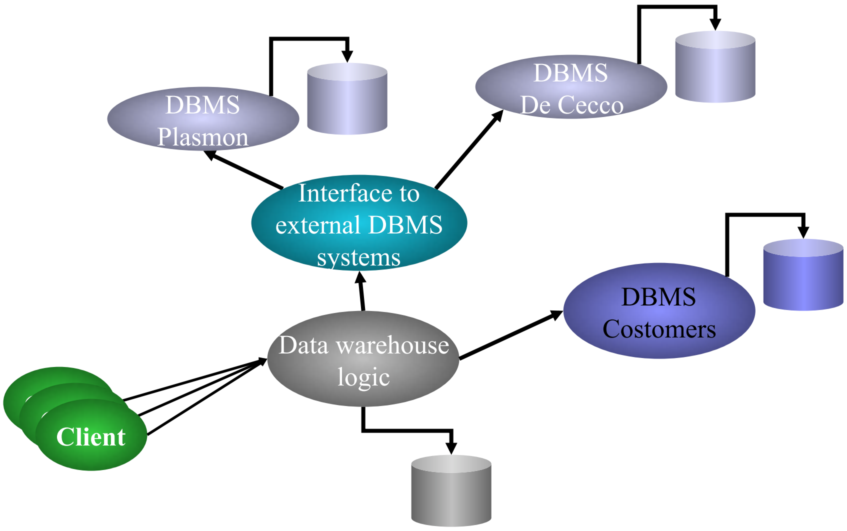
- In the typical scheme, the mid level has all the application logic
- Advantage
  - decoupling of logic and data, logic and presentation

An example

Data warehouse for a supermarket

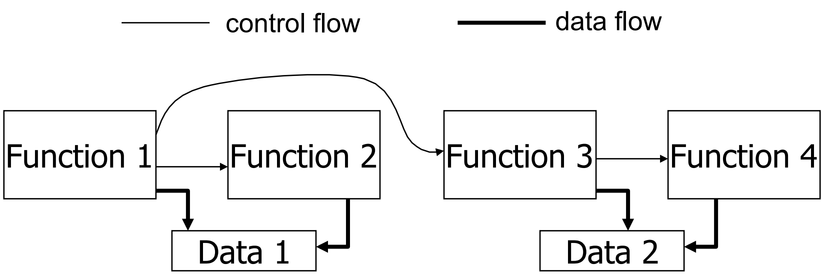


From 3 to n levels

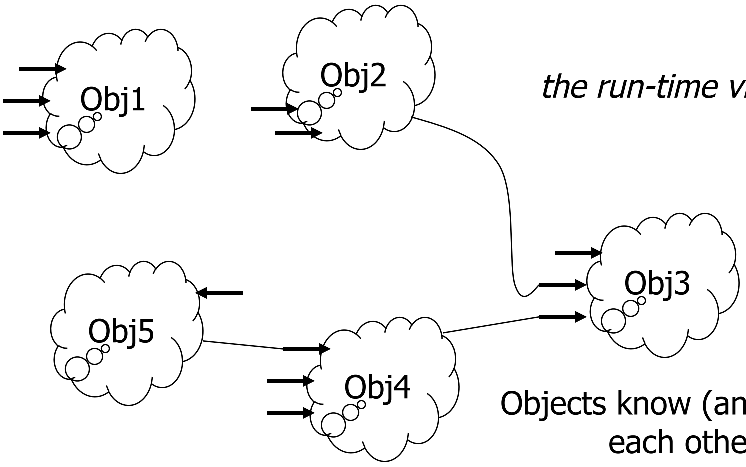


A functional architecture

- The system is decomposed into abstract operations
- Operations know (and name) each other
- Connectors = operation call/return
- Additional connectors via shared data



An object-oriented architecture



the run-time view

Objects know (and name) each other

Kinds of objects

- Active vs. passive
  - different coordination models possible
- Local vs. distributed
  - References can be local or remote
  - Activation / deactivation can be local or remote
    - parameter passing
  - Local vs. global guarantees (latency, reliability, security, …)
  - Migration
- Persistent vs. volatile

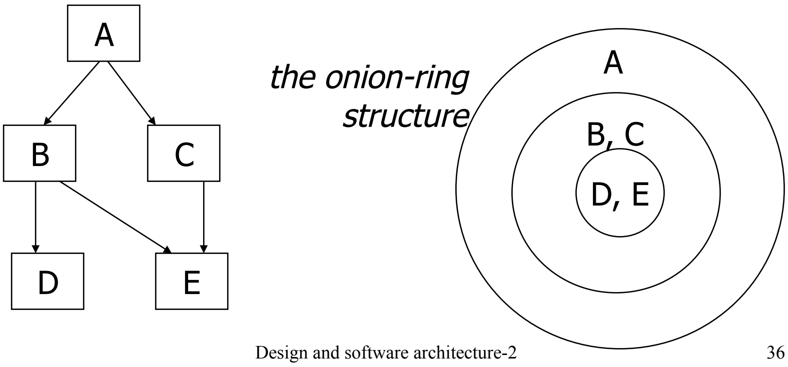
Distributed objects (3)

- Stateless vs. statefull objects
- Statefull objects have to save their state between
  - object deactivation and
  - object activation

  onto persistent storage
- Can be achieved by
  - externalization into file system
  - mapping to relational database
  - object-oriented database

Layered system

- The system is organized through abstraction levels, as a hierarchy of abstract machines
- Hierarchy is given by the USE relation



Pipes&filter style

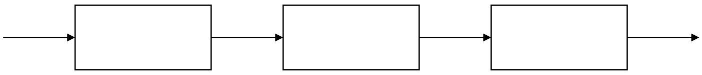
filters

The "Unix" model

pipes

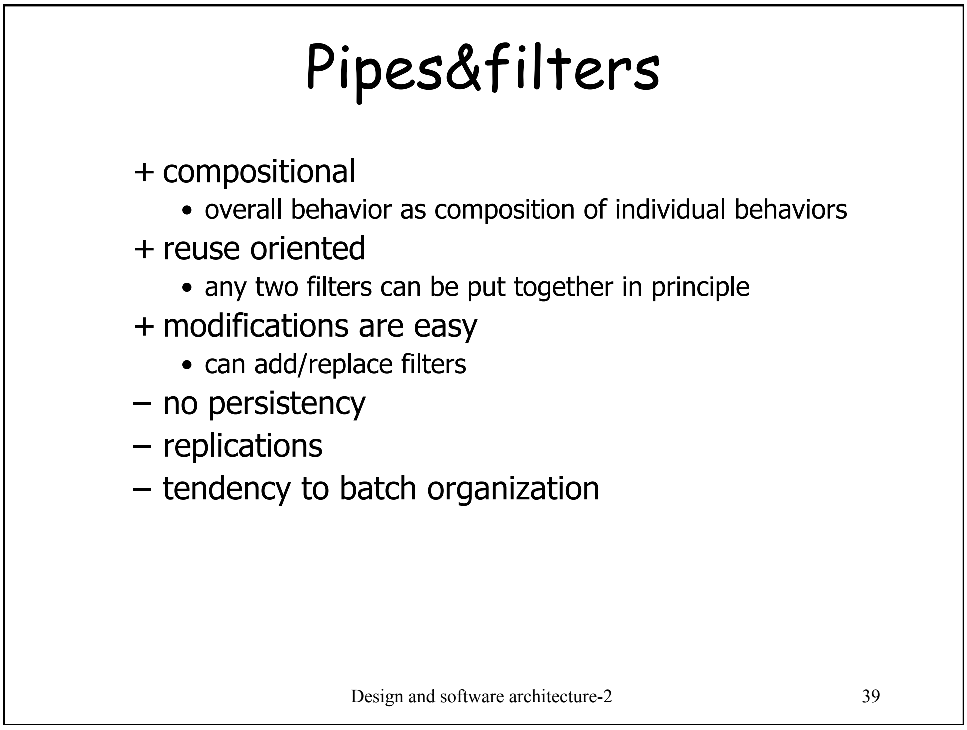
Filters get and put data on their input and output pipes; they ignore the existence (and identity) of other filters.

Pipes&filters



This is a pipeline

- Various control regimes are possible
  - sequential batch
    - e.g., a batch multi-pass compiler
  - concurrent



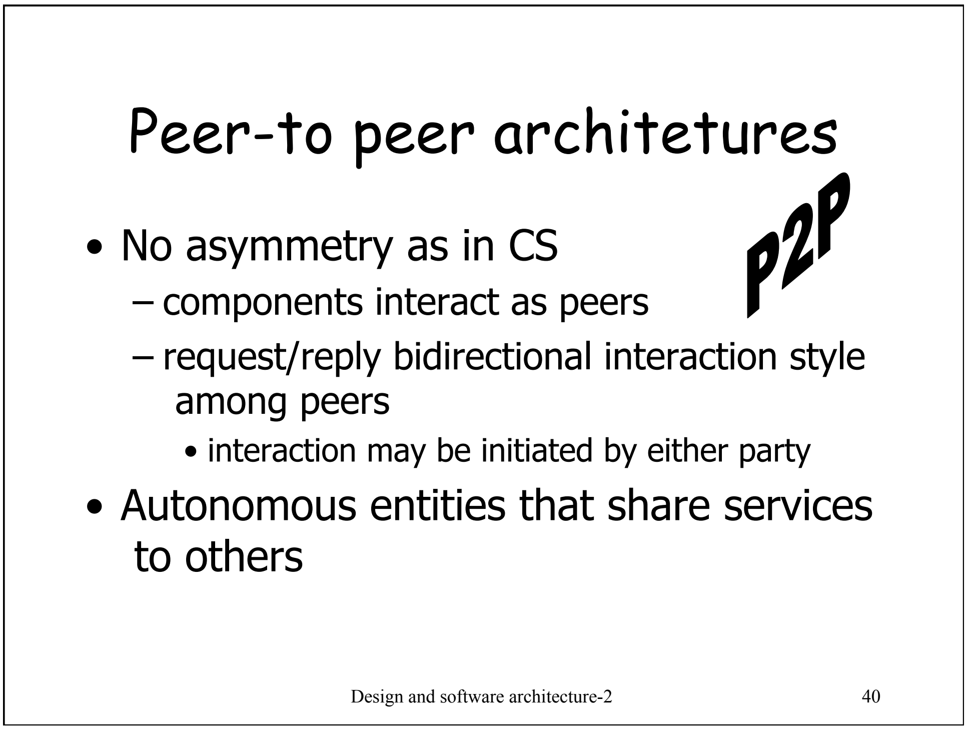
Pipes&filters

+ compositional
    - overall behavior as composition of individual behaviors
+ reuse oriented
    - any two filters can be put together in principle
+ modifications are easy
    - can add/replace filters

– no persistency

– replications

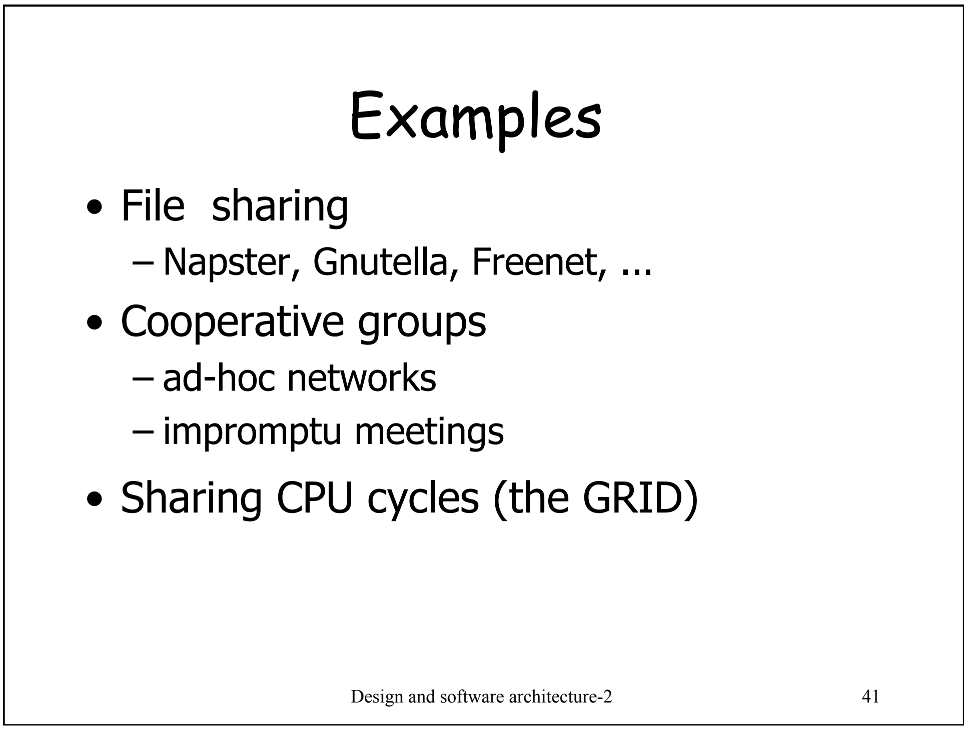
– tendency to batch organization



Peer-to peer architetures

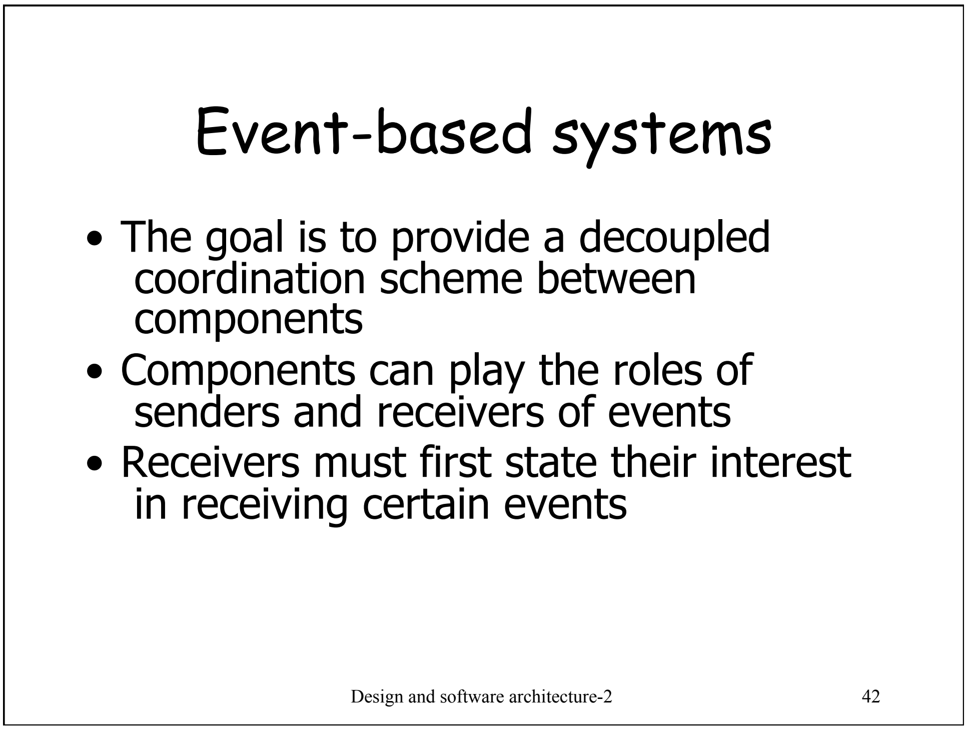
P2P

- No asymmetry as in CS
    - – components interact as peers
    - – request/reply bidirectional interaction style among peers
        - interaction may be initiated by either party
- Autonomous entities that share services to others



Examples

- File sharing
  - Napster, Gnutella, Freenet, ...
- Cooperative groups
  - ad-hoc networks
  - impromptu meetings
- Sharing CPU cycles (the GRID)



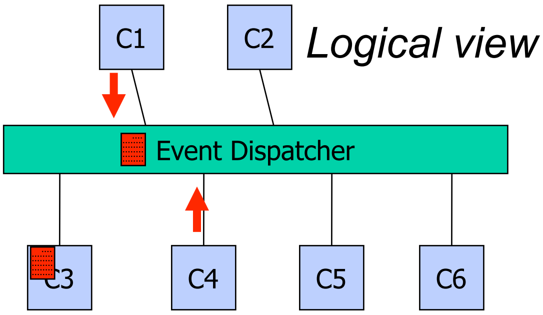
Event-based systems

- The goal is to provide a decoupled coordination scheme between components
- Components can play the roles of senders and receivers of events
- Receivers must first state their interest in receiving certain events

Taxonomy of event-based systems

- Listener mode: observer pattern
  - receivers register with senders
  - sender/receiver communication is point-to-point
- Database trigger mode
  - "senders" update tables
  - "receivers" (or internal code) notified of changes
- Broker mode
  - receivers register with common message broker
  - sender/receiver communication is mediated by message broker through a *middleware service*

Broker-mode event-based systems (publish/subscribe)



C1

C2

Logical view

Event Dispatcher

C3

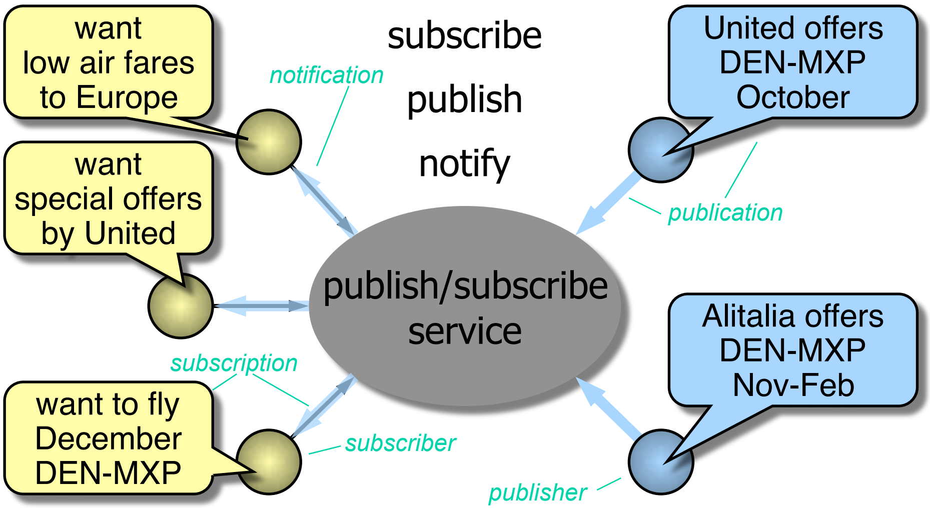
C4

C5

C6

Publish/Subscribe Services

Example due to Carzaniga and Wolf



want low air fares to Europe

want special offers by United

want to fly December DEN-MXP

subscribe

publish

notify

publish/subscribe service

United offers DEN-MXP October

Alitalia offers DEN-MXP Nov-Feb

notification

subscription

subscriber

publication

publisher

Features (1)

- Publish
  - event generation
- Subscribe
  - declaration of interest
- Events are broadcast to all registered components
- No explicit naming of target component
- Different kinds of guarantees possible

Features (2)

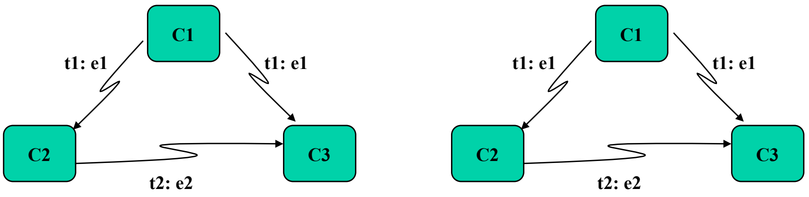
- + Increasingly used for modern
  - + *Widely used as "listener mode" for user interfaces*
- + Easy integration strategies
- + Easy addition/deletion of components
- – Potential scalability problems
- – Ordering of events

Features (3)

- Asynchrony
  - – send and forget
- Reactive
  - – computation driven by receipt of message
- Location/identity abstraction
  - – destination determined by receiver, not sender
- Loose coupling
  - – senders/receivers added without reconfiguration
  - – one-to-many, many-to-one, many-to-many

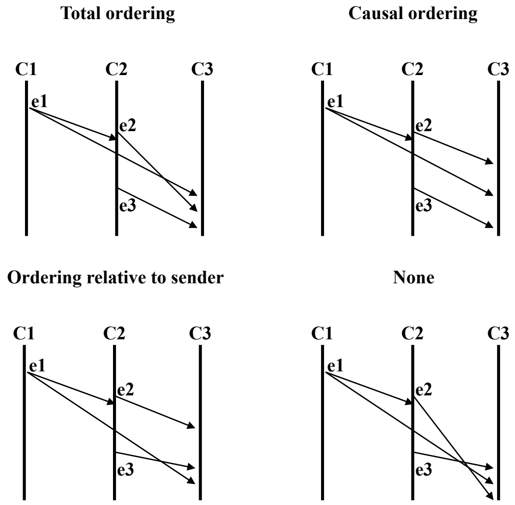
Critical aspects – ordering of events

- PS adopts an asynchronous communication model
  - Ordering of events (handling) is crucial



Ordering of events

<u>Hypothesis:</u>
e3 generated by C2 as a consequence of receipt of e1


Total ordering

Causal ordering

Ordering relative to sender

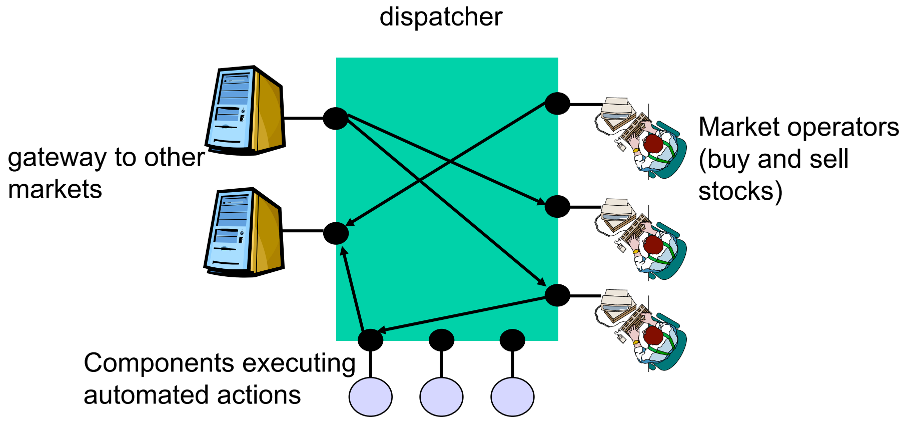
None

More critical aspects

- Possible delivery guarantees
  - Best effort
  - At least once
  - At most once
  - Once and only once

Banking and e-commerce systems

- TIBCO, ION-MkView, smartsockets



dispatcher

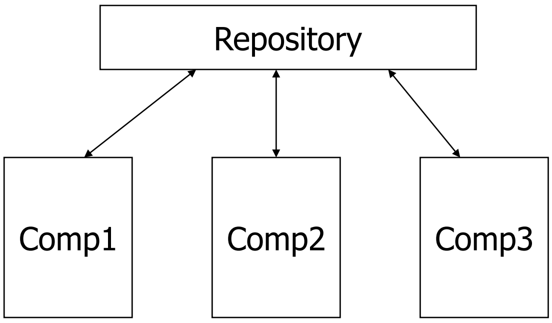
gateway to other markets

Market operators (buy and sell stocks)

Components executing automated actions

Repository-based systems

- Components communicate only through a repository


Repository

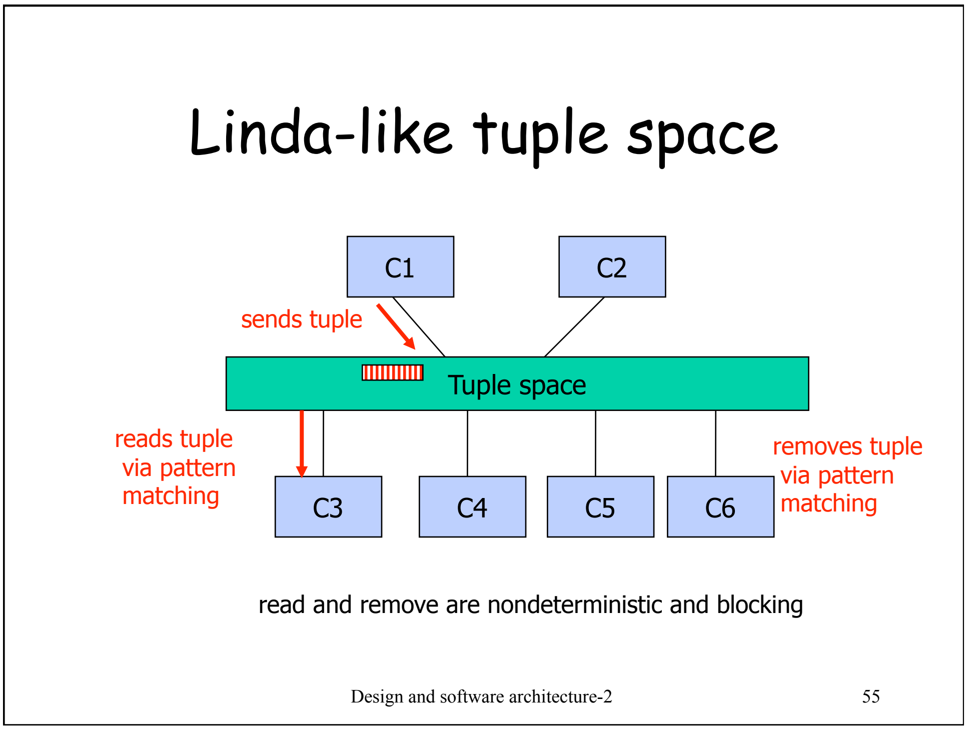
Comp1

Comp2

Comp3

The style

- Components are active; repository is passive
- Repository is persistent (on the opposite, events are transient)
- Sometimes several actions are grouped in a *transaction*



Linda-like tuple space

C1

C2

sends tuple

Tuple space

reads tuple via pattern matching

C3

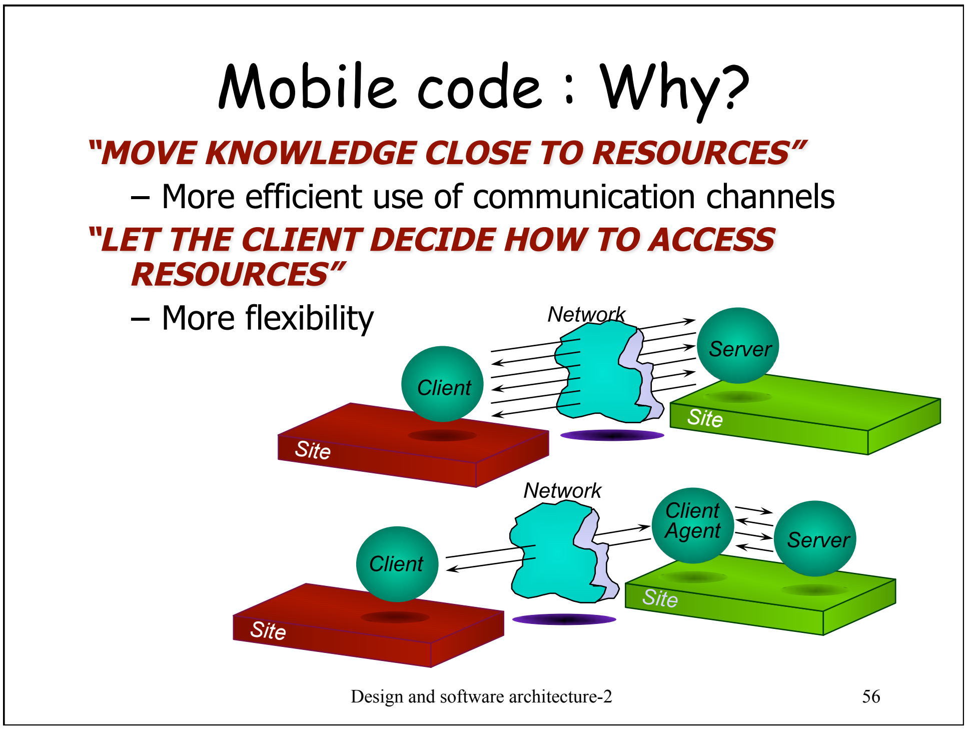
C4

C5

C6

removes tuple via pattern matching

read and remove are nondeterministic and blocking



Mobile code : Why?

"MOVE KNOWLEDGE CLOSE TO RESOURCES"

– More efficient use of communication channels

"LET THE CLIENT DECIDE HOW TO ACCESS RESOURCES"

– More flexibility

Network

Client

Server

Site

Site

Network

Client

Client Agent

Server

Site

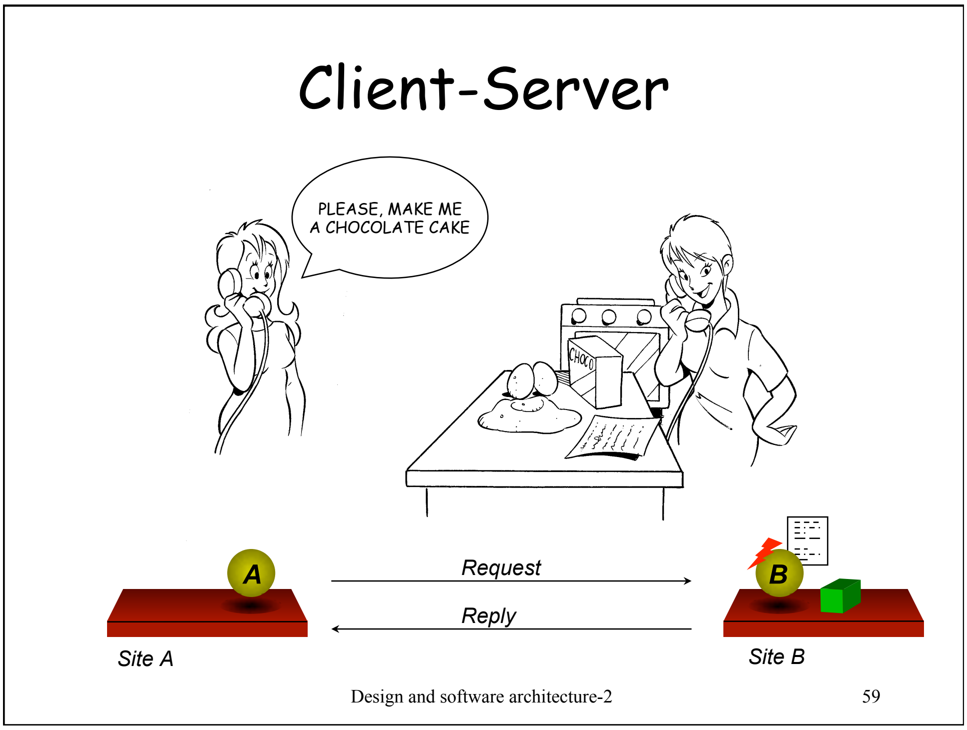
Site

Different architectural styles

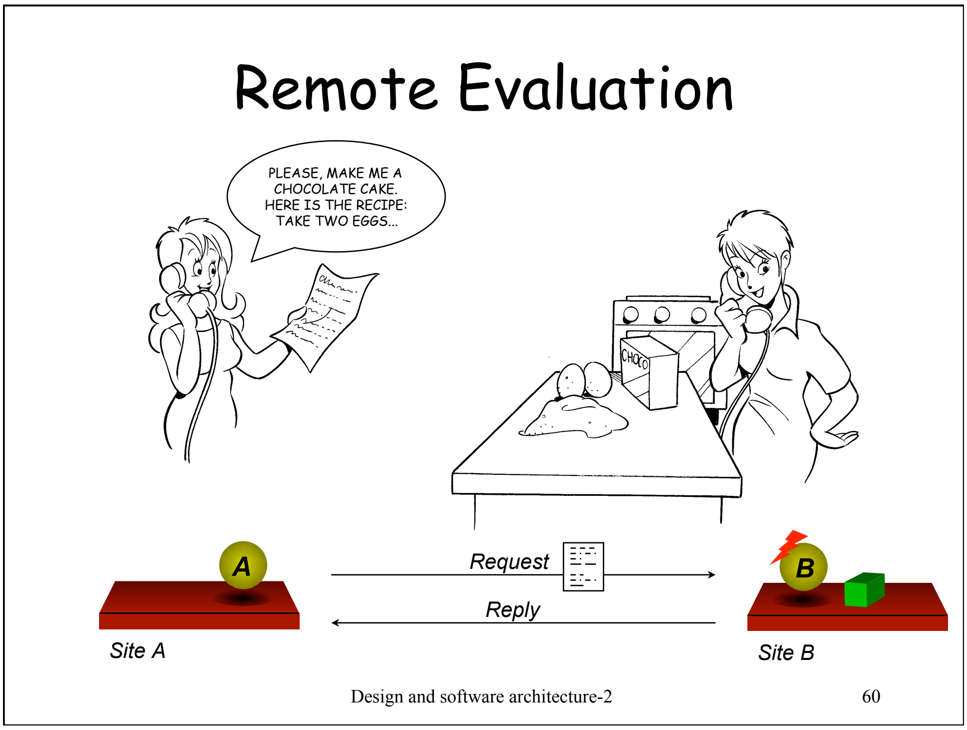
- In general, to achieve a service, one needs:
  - code necessary to execute the service
  - resources
  - engine to execute it
- Different "location modes" define different architectural styles

An example--How to make a cake

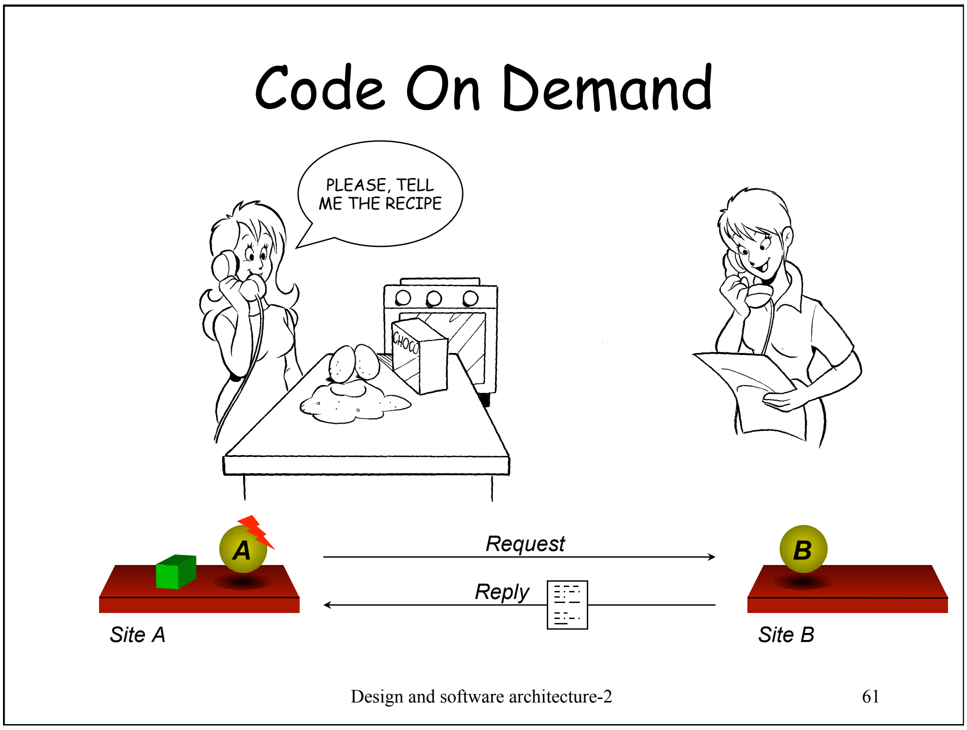
Example due to Carzaniga, Picco, Vigna
Designing Distributed Applications with Mobile Code Paradigms
ICSE ‘97



Client-Server



Remote Evaluation



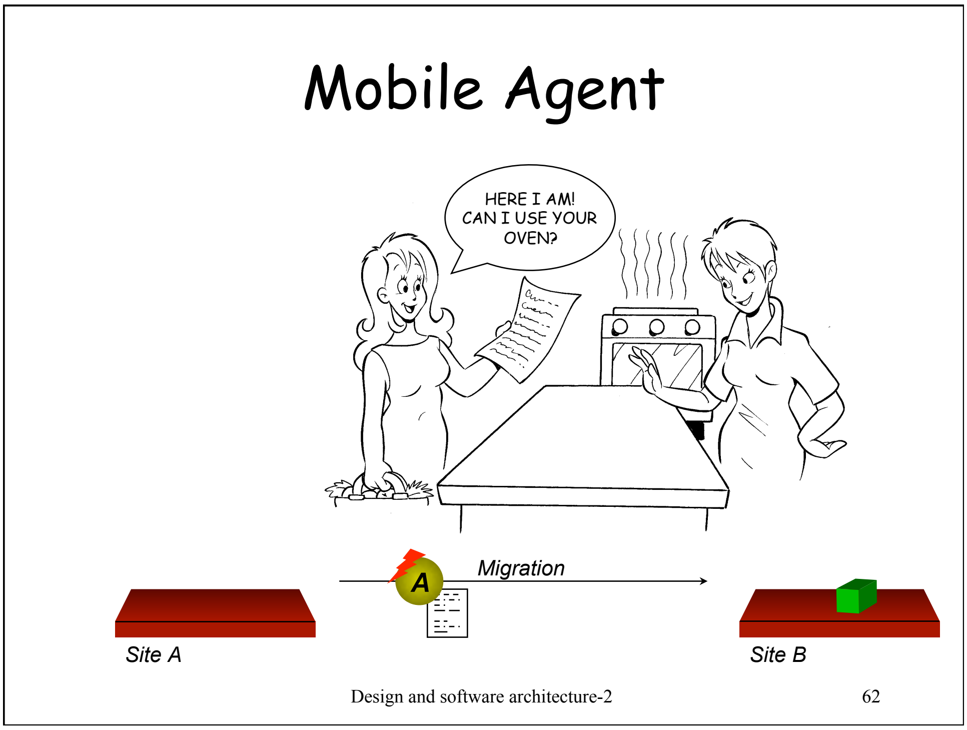
Code On Demand

Request

Reply

Site A

Site B



Mobile Agent

Migration

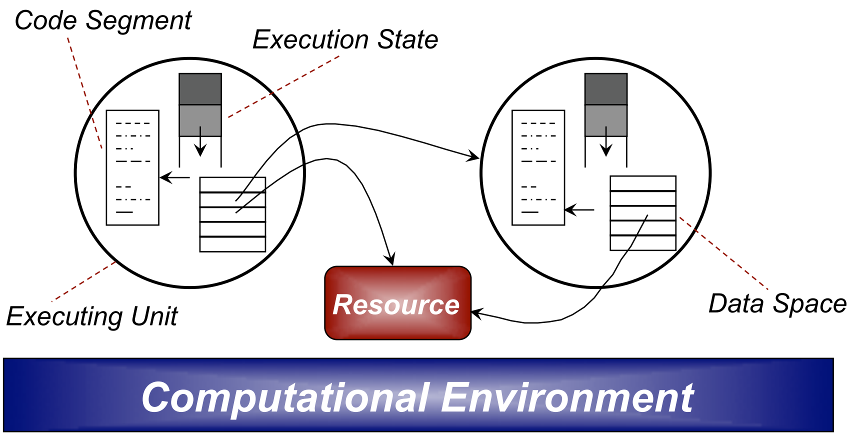
Site A

Site B

Mobile code

- Location is visible
  - it is a concept to be exploited at design time
- Distributed application is a set of nodes (*computational environments*)
  - providing support to execution of mobile components
  - supporting access to resources
- Components may migrate from node to node
- Their behavior may change because code may be dynamically loaded from other nodes

A reference model

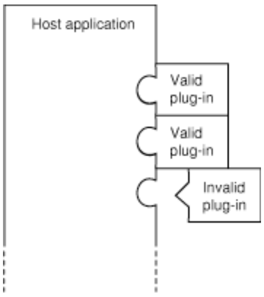


Two notions

- ***Strong*** mobility
  - code & state migrate from an executing unit to a new computational environment
- ***Weak*** mobility
  - code can migrate among computational environments

The plug-in style

- *Plug-in*: a code fragment that adds functionality to an application, the *host application*
- *Host application:* offers the mechanisms to add new plug-ins during operation
- Plug-ins have to comply to the extensibility constraints defined in architecture of the application
- Examples
  - Adobe Photoshop graphic filters
  - Emacs
  - Eclipse



Advantages for application developers

- You can implement and incorporate application features very quickly
- Because plug-ins are separate modules with well-defined interfaces, you can quickly isolate and solve problems
- You can create custom versions of an application without source code modifications
- Third parties can develop additional features
- Plug-in interfaces can be used to wrap legacy code written in different languages

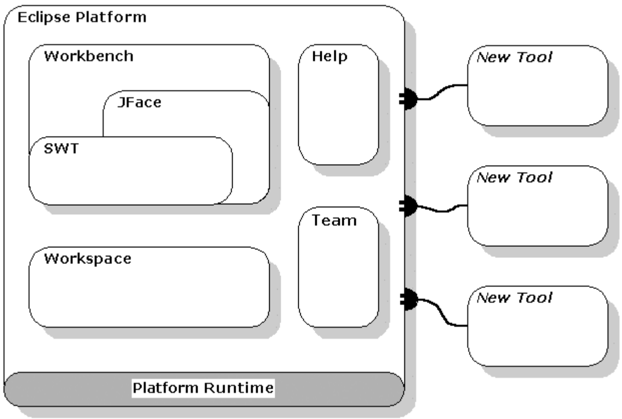
Eclipse case-study

- Extensible platform for building Interactive Development Environments (IDEs)
- Provides core services for controlling tools working together
- Tools are developed as Eclipse plug-in
- Can be seen as a runtime platform to support system development by composition of plug-ins
- The Eclipse layers (from the bottom)
  - The platform
  - The Java development tools (JDT)
  - Plug-in development environment (PDE)

http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html

http://www.cs.queensu.ca/home/stl/pdf/eclipse_plugins_1up.pdf

Eclipse architecture



SWT: graphics and standard set of widgets
JFace: common UI tasks; higher-level toolkit built on top of SWT

Eclipse plug-ins

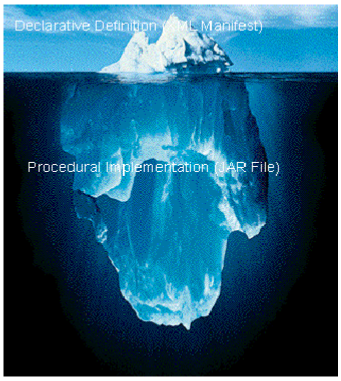
- All written in Java
- Found at Eclipse launch
  - Heavier-weight than Emacs model – can't dynamically swap plug-ins or write and execute code
- Include a "manifest" file
  - Specifies visibility of included classes and methods
  - Used to add information to the plug-in registry

Manifest file example

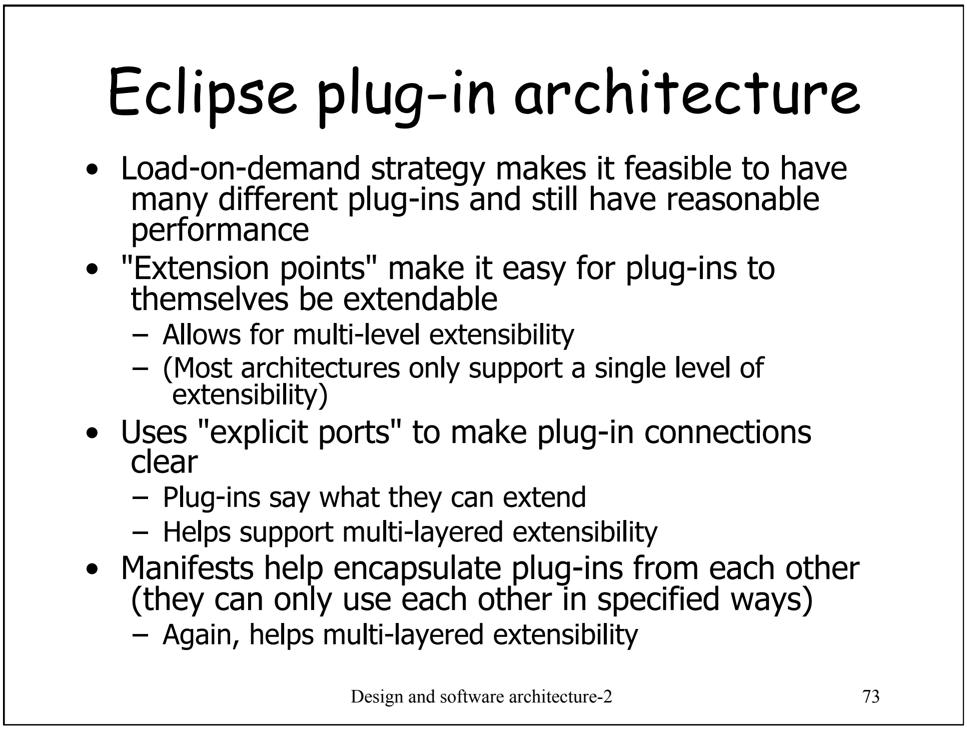
```
•  <?xml version="1.0" encoding="UTF-8"?>
•  <plugin
•    name="JUnit Testing Framework"
•    id="org.junit"
•    version="3.7"

•    provider-name="Eclipse.org">
•    <runtime>
•      <library name="junit.jar">
•        <export name="*"/>
•      </library>
•    </runtime>
•  </plugin>
```

A view

- Distinction between declaration (in XML) and implementation (Java)
- Lazy loading of plug-ins: performed at deployment time

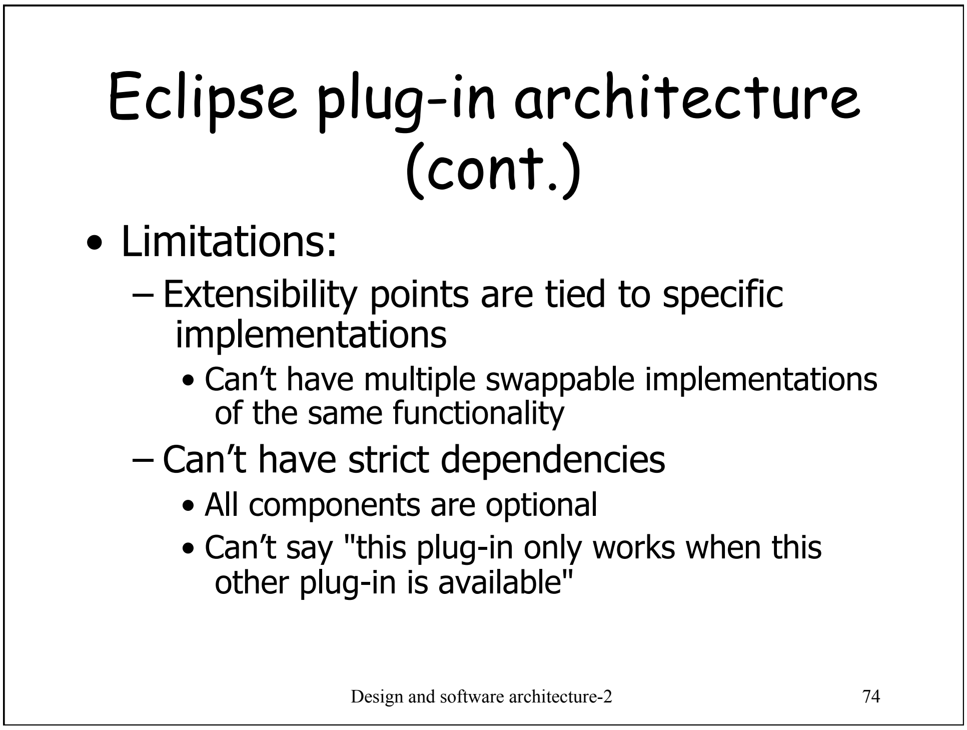

Declarative Definition (XML Manifest)
Procedural Implementation (JAR File)

from http://www.cs.queensu.ca/home/stl/pdf/eclipse_plugins_1up.pdf



Eclipse plug-in architecture

- Load-on-demand strategy makes it feasible to have many different plug-ins and still have reasonable performance
- "Extension points" make it easy for plug-ins to themselves be extendable
  - Allows for multi-level extensibility
  - (Most architectures only support a single level of extensibility)
- Uses "explicit ports" to make plug-in connections clear
  - Plug-ins say what they can extend
  - Helps support multi-layered extensibility
- Manifests help encapsulate plug-ins from each other (they can only use each other in specified ways)
  - Again, helps multi-layered extensibility

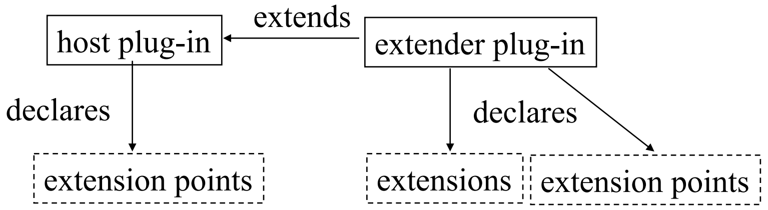


Eclipse plug-in architecture (cont.)

- Limitations:
  - Extensibility points are tied to specific implementations
    - Can’t have multiple swappable implementations of the same functionality
  - Can’t have strict dependencies
    - All components are optional
    - Can’t say "this plug-in only works when this other plug-in is available"

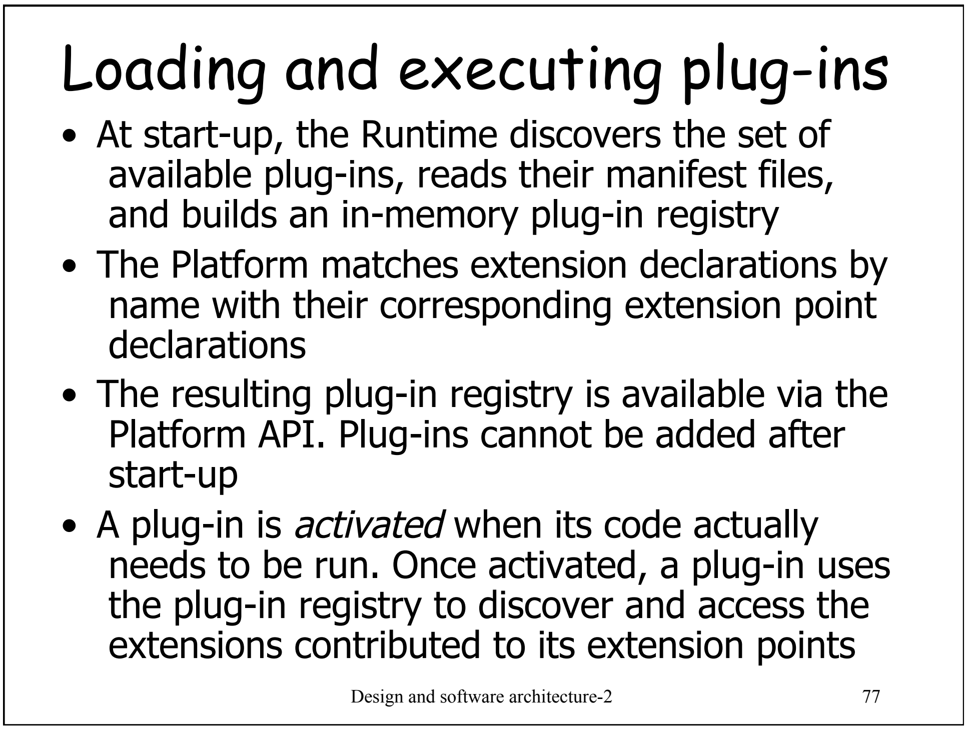
Declaration of plug-in

- Manifest file specifies the interconnections with the external world in terms of
  - Extension points
  - Extensions to one or more extension points defined in other plug-ins



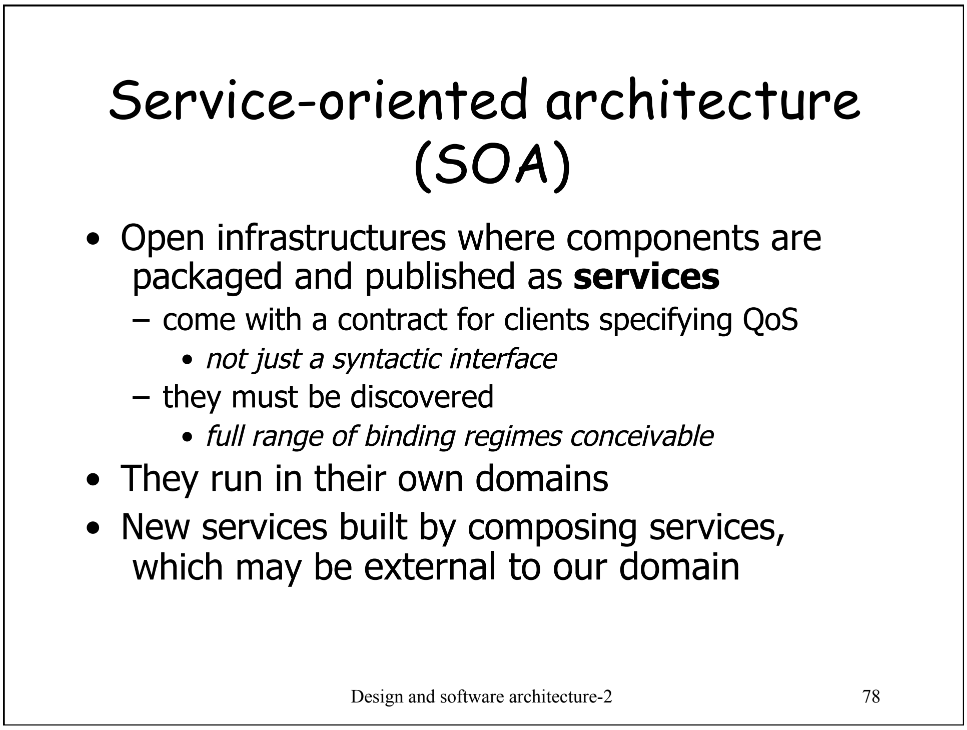
Extension of plug-ins

- A Java interface is associated with an extension point by the host plug-in developer
- The extender plug-in has to provide a call -back object that implements such interface
- This enables the host plug-in to interact with the extender plug-in
- Interaction in the other direction (from the extender to the host plug-in) is implementation dependent



Loading and executing plug-ins

- At start-up, the Runtime discovers the set of available plug-ins, reads their manifest files, and builds an in-memory plug-in registry
- The Platform matches extension declarations by name with their corresponding extension point declarations
- The resulting plug-in registry is available via the Platform API. Plug-ins cannot be added after start-up
- A plug-in is *activated* when its code actually needs to be run. Once activated, a plug-in uses the plug-in registry to discover and access the extensions contributed to its extension points



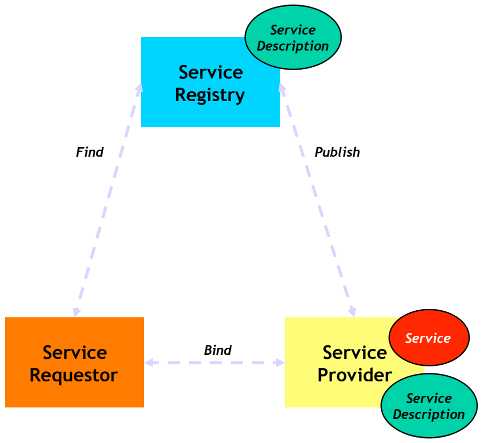
Service-oriented architecture (SOA)

- Open infrastructures where components are packaged and published as **services**
  - come with a contract for clients specifying QoS
    - *not just a syntactic interface*
  - they must be discovered
    - *full range of binding regimes conceivable*
- They run in their own domains
- New services built by composing services, which may be external to our domain

SOAs: roles

- A SOA defines three roles
  - **Service requestors** request the execution of a service
  - **Service providers** they offer services
  - **Intermediaries** (service registries): they provide information (metadata) about other information /services
- The interaction among these roles results in the execution of the following operations
  - Service description publication
  - Service discovery
  - Service binding

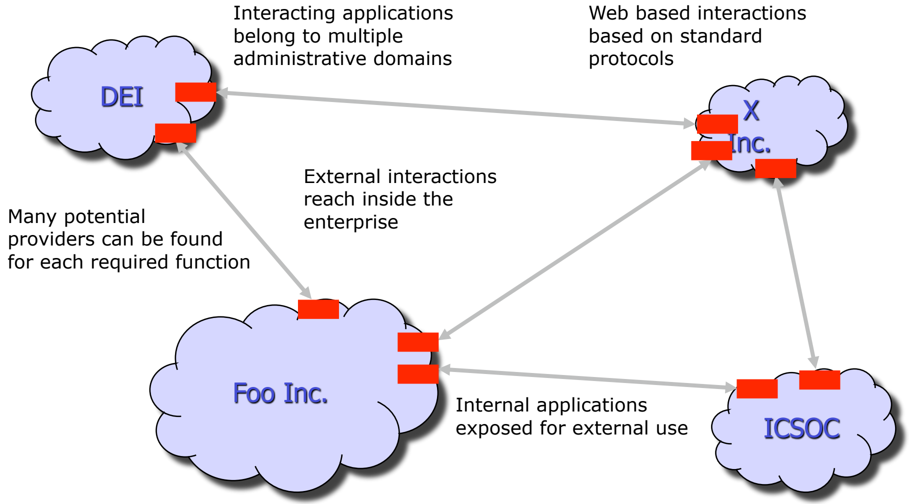
Roles and operations



The case of WEB services

- Extend the WEB paradigm to searching application (services), not just passive information
- Use standard protocols for data exchange (XML, SOAP)--see later

WEB services and networked enterprises



Interacting applications belong to multiple administrative domains

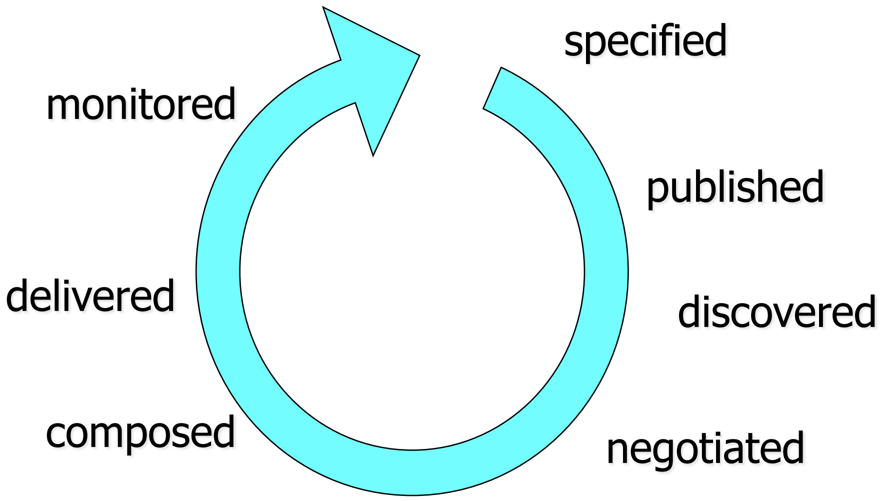
Web based interactions based on standard protocols

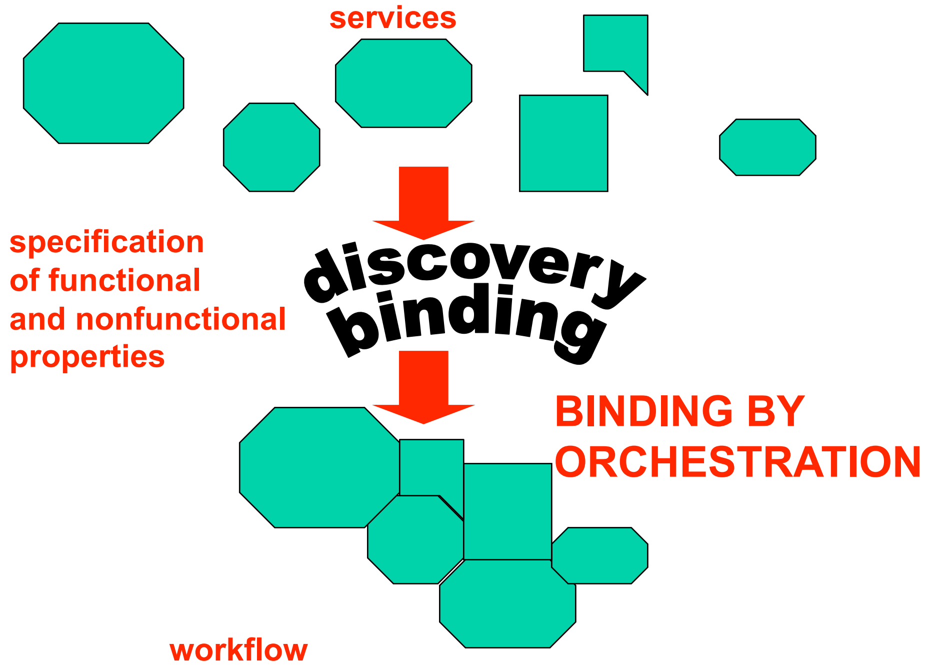
External interactions reach inside the enterprise

Many potential providers can be found for each required function

Internal applications exposed for external use

Service lifecycle





Open issues

- Security
- Reliability of message exchange
- Quality of Service
- Service maintainability
- Transaction management
- Service composition
  - Services collaborate to offer a more advanced service

Software architecture and middleware

Middleware

- Provides services to program a component-based, distributed system, as extensions of the operating system
- It is a layer on top of the OS, used by applications
- Enforces a certain style

RMI as an example

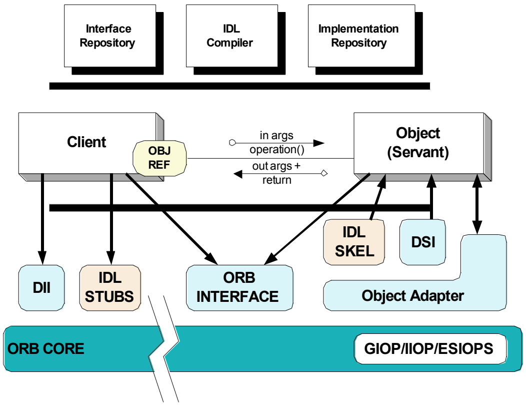
- Part of a language (Java)
- Support localization of remote objects (registry)
- Supports communication with remote objects in a client-server style
  - Changes in semantics of params
- Supports dynamic class loading

CORBA

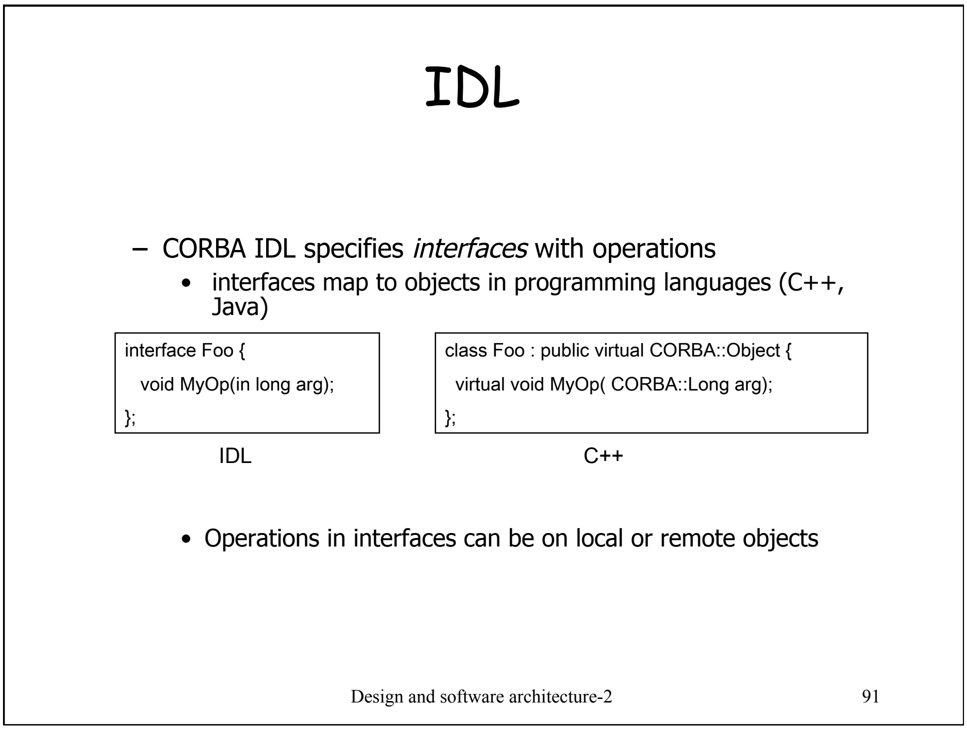
- Common Object Request Broker Architecture (CORBA) is an open distributed object computing infrastructure standardized by the Object Management Group

Overview of CORBA

- CORBA shields applications from heterogeneous platform *dependencies*
  - *e.g.,* languages, operating systems, networking protocols, hardware



- It simplifies development of distributed applications by automating/encapsulating
  - Object location
  - Connection & memory mgmt.
  - Parameter (de)marshaling
  - Event & request demultiplexing
  - Error handling & fault tolerance
  - Object/server activation
  - Concurrency
  - Security
- CORBA defines *interfaces*, not *implementations*



IDL

- CORBA IDL specifies *interfaces* with operations
  - interfaces map to objects in programming languages (C++, Java)

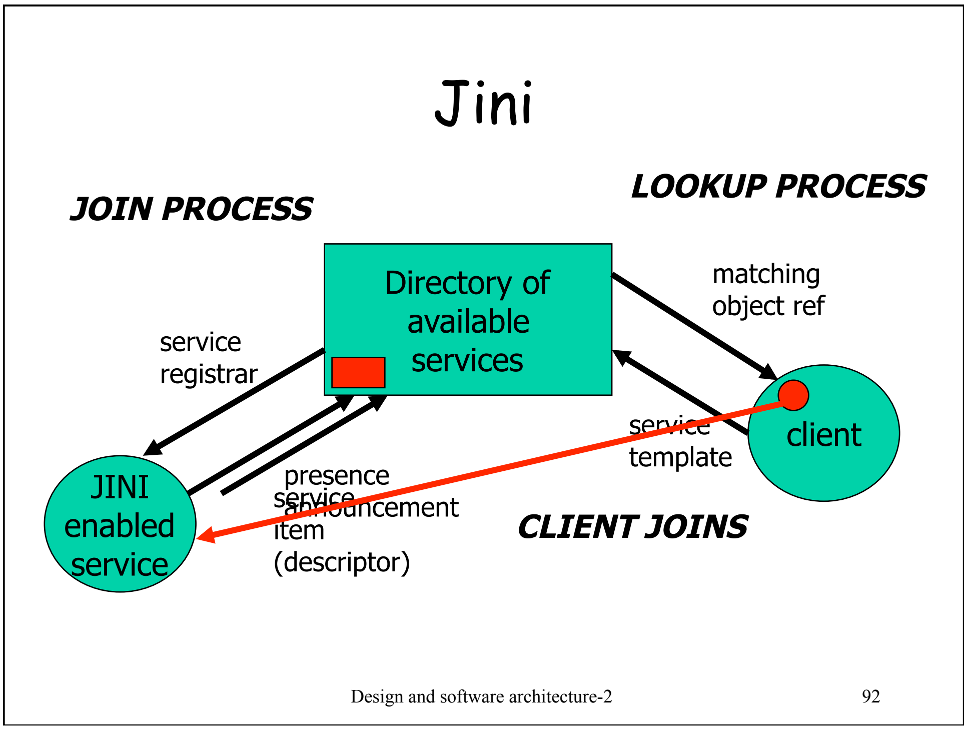
```
interface Foo {
   void MyOp(in long arg);
};
```

IDL

```
class Foo : public virtual CORBA::Object {
  virtual void MyOp( CORBA::Long arg);
};
```

C++

- Operations in interfaces can be on local or remote objects



Jini

JOIN PROCESS

LOOKUP PROCESS

Directory of available services

service registrar

matching object ref

service template

client

JINI enabled service

presence announcement

service item (descriptor)

CLIENT JOINS

Gnutella: a P2P middleware

- An open source software providing support to bidirectional info transfer among peers
- a peer communicates directly with a small number of friendly sites or with public Gnutella sites
- Gnutella provides protocols for
  - discovery
  - information search and exchange

SOAP, an XML-based approach

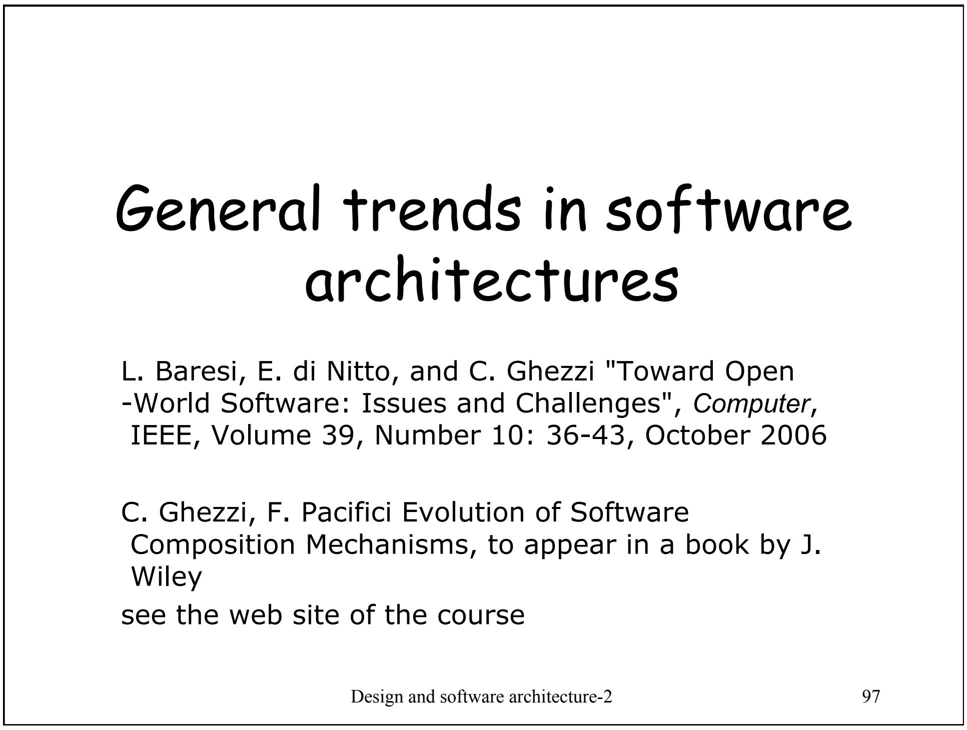
- Remote Procedure Call protocol that works over the Internet
- A message is written in XML and it usually sent as an HTTP-POST request
  - a procedure executes on the server and the value it returns is also formatted in XML
  - parameters can also be complex records and lists
- Does not specify APIs or how calls should be managed (serially, in parallel …)
- It is the basis for SOA approaches

Middleware and application server

- A new middleware generation
- Which deal not only with component interactions
- But also define a specific model for components
- Provide a declarative way of changing components and their interactions

Application Server

- A possible definition
  - Software layer on top the operating system providing a runtime infrastructure and a set of functionalities to support **integration**, **deployment** and **execution** of applications and server components in a distributed multi-tier architecture
- However, there is no universally accepted definition!
- You will see J2EE separately



General trends in software architectures

L. Baresi, E. di Nitto, and C. Ghezzi "Toward Open -World Software: Issues and Challenges", *Computer*, IEEE, Volume 39, Number 10: 36-43, October 2006

C. Ghezzi, F. Pacifici Evolution of Software Composition Mechanisms, to appear in a book by J. Wiley

see the web site of the course