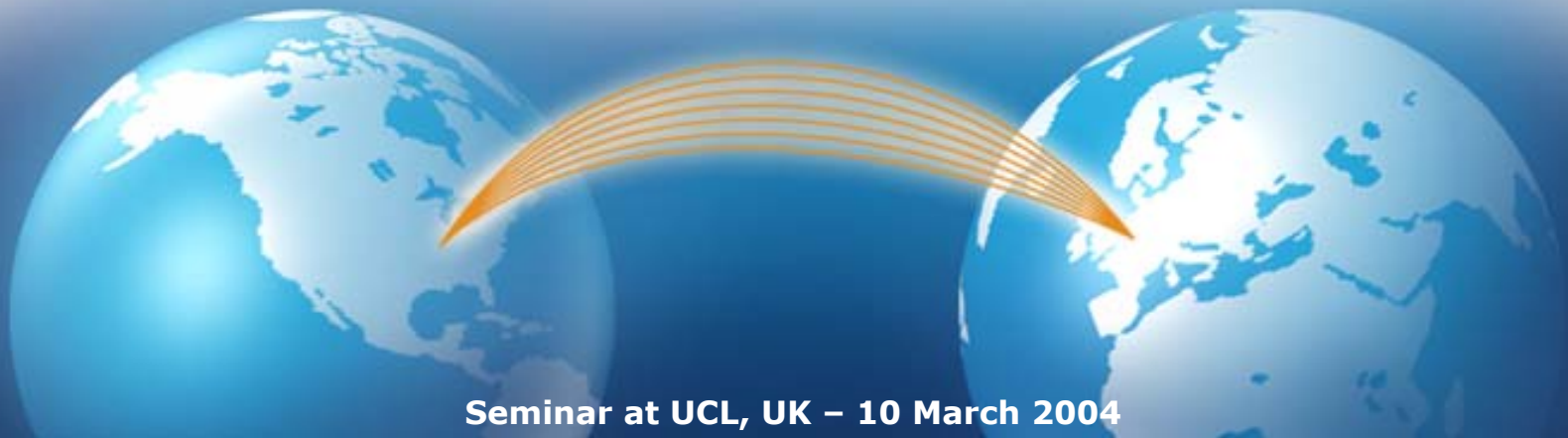
Perspectives on Software Architecture

J.P. Martin-Flatin, CERN, Switzerland
jp.martin-flatin@ieee.org
http://cern.ch/jpmf/



Seminar at UCL, UK – 10 March 2004

Outline

- Motivation for a new research discipline
- Introduction to software architecture
- Examples
- More on software architecture
- Research perspectives

Motivation for a new Research Discipline

Problems with Software (1/2)

- Most software engineers waste time and money because they keep reinventing the wheel:
  - most applications are built almost from scratch
- Generation after generation, most software engineers make the same mistakes over and over again:
  - as a community, we don't seem to learn from our mistakes
- Most software systems out there are difficult to maintain
- Software projects are often expensive:
  - Almost impossible to do something decent at a low price

Problems with Software (2/2)

- Too many software projects fail:
  - Budget overrun
  - Missed deadlines
  - Projects killed before software delivered to users (customers)
- Today's software requires more flexibility than ever before:
  - Technology moves faster (languages, middleware, components, protocols, etc.)
  - Large investments still have long-term payback
  - Most applications written today will need to survive major technological changes

Solution Space: Two Dimensions (1/2)

- Software engineering:
  - Better design:
    - Encourage it (education)
    - Facilitate it (engineering)
  - More reuse:
    - Encourage it (education)
    - Facilitate it (engineering)
  - Better requirements
  - Better coding
  - Better languages
  - Better tools
  - etc.

Software architecture

Solution Space: Two Dimensions (2/2)

- Finance:
  - How to charge development cost of reusable software?
  - How to assess TCO of software?
  - See Hohmann's book

Better Design (1/2)

- Some designs facilitate reuse
- We can learn from our mistakes:
  - Study and improve poor designs
  - Study good designs (best practices)
- Maintenance-oriented design and coding:
  - Think maintenance
  - Anticipate change
  - Flexibility (indirections)

Better Design (2/2)

- Impact of design quality on software project costs:
  - During 1st iteration of an iterative software development process:
    - Increases design cost:
      - More time
      - Smarter people
    - Unpredictable effect on implementation cost:
      - Some projects can get by with quick & dirty coding
    - Decreases debugging cost
    - Decreases testing cost
  - During further iterations:
    - Decreases all costs
  - Always decreases maintenance cost

Reuse (1/2)

- Stop reinventing the wheel:
  - Develop once, reuse code many times
- Stop making the same mistakes
- Leverage best practices:
  - Specify design once, reuse design many times
- Facilitate maintenance

Reuse (2/2)

- Impact of reuse on software project costs:
  - During 1st iteration of an iterative software development process:
    - Unpredictable effect on design cost:
      - How much do we reuse?
      - How many reusable designs do we generate?
    - Unpredictable effect on implementation cost
    - Decreases debugging cost
    - Decreases testing cost
  - During further iterations:
    - Decreases all costs
  - Always decreases maintenance cost

Designing Complex Applications (1/2)

- OO software development process:
  - No standard:
    - Booch/OOAD, OMT, OOSE, Catalysis, RUP
  - Five phases:
    - Requirements + analysis (what)
    - High-level design (how-to, big picture)
    - Low-level design (detailed how-to)
    - Implementation
    - Testing + deployment
  - Multiple iterations

Designing Complex Applications (2/2)

- Software architecture:
  - During high-level design phase (big picture)
  - Functional aspects vs. non-functional aspects
  - Disentangle independent aspects
  - Framework for dependencies between subsystems:
    - E.g. if we adopt a service-oriented architecture:
      - Migration from distributed objects to services
      - From tightly coupled to loosely coupled subsystems
  - Focus: *features and capabilities*

Introduction to Software Architecture

Software Architecture: Definitions (1/2)

- "Art of structuring complex applications properly."
- "The nightmare of stock markets: a long-term investment in the ability of software to evolve without immediate ROI."

Software Architecture: Definitions (2/2)

- Shaw & Garlan:
  "As the size and complexity of software systems increase, the design and specification of overall system structure become more significant issues than the choice of algorithms and data structures of computation. Structural issues include the organization of a system as a composition of components; global control structures; the protocols for communication, synchronization, and data access; the assignment of functionality to design elements; the composition of design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives. This is the *software architecture* level of design."

Main Concerns in Software Architecture

- Distribution
- Scalability:
  - Bottlenecks
  - Performance
- Flexibility:
  - *At a given time:* heterogeneity, interoperability, portability
  - *Over time:* evolvability, modifiability
- Robustness
- Security
- Synchronization
- Data access
- Integration

Is It New?

- No:
  - Experienced software engineers have been doing this implicitly for years
- Yes:
  - Documenting best practices is new
  - Teaching best practices is new
  - Learning from others' mistakes is new
  - Using complex software everywhere is new:
    - Used to be confined to scientific community
  - Designing very flexible software is new

What Should Software Architects Do?

- Features:
  - Fulfill requirements in due time within budget
- Capabilities:
  - Anticipate change:
    - Prepare for e-business
  - Make trade-offs
  - Isolate independent concerns

Anticipate Change (1/2)

- Among all the subsystems that constitute my application, where do I expect to add more functionality in the future?
  - e.g., plug-ins for Web browsers
- What new demands do I expect from my users?
- Will my software always run on the same OS, using the same middleware and the same component technologies?
- Will the communication protocols change?
- Will the data format change?
- Will the database technology change?

Anticipate Change (2/2)

- Will the security requirements or technologies change?
- Which subsystems do I need to change if the load increases 10, 100 or 1000 times?
  - Where are the bottlenecks of my software?
- Which subsystems do I need to change to make my application fault tolerant?
  - Do I have to break everything?
- How long will it take a new developer to understand the architecture of my software?

Examples of Anticipated Changes (1/2)

- So you thought these were details?
  - Y2K
  - VAT rates
  - International unit system vs. U.S. units
  - Key length for SSL encryption
- Java vs. C#:
  - Tightly coupled distributed systems: dead?
- Components:
  - CORBA/CCM vs. J2EE vs. .NET vs. ???

Examples of Anticipated Changes (2/2)

- Service discovery:
  - Web Services vs. CORBA vs. LDAP vs. Jini vs. ???
  - Via traders?
- Data access:
  - RDBMS vs. OODBMS vs. KBMS
- Application-domain specific communication protocols vs. one-size-fits-all HTTP
- Application-domain specific data representation/encoding vs. XML
- In hospitals, patient files will include image formats that do not exist today (MPEGx)

Prepare for E-Business

- Software written today should prepare for B2B/B2C:
  - Wherever we require interactive input, we should allow for XML-based input
  - Tomorrow XML may be replaced with another technology
- E.g., online retail software:
  - Most subsystems should ignore how orders are placed
  - Orders may be taken by:
    - Frontdesk officer
    - Telephone
    - Web (B2B, B2C)
    - Letter
    - Fax
    - etc.

Make Trade-Offs (1/2)

- Which of my subsystems need to be tightly coupled? Which ones need to be loosely coupled?
- Will administrators control my application via:
  - high-level configuration mechanisms (policies, goals)
  - low-level ones (CLI)
- Should the physical distribution of my subsystems be totally free or partially constrained?

Make Trade-Offs (2/2)

- Business aspects that influence architecture:
  - For which subsystems should I purchase COTS software? For which ones do I need ad hoc developments?
  - Initial development costs vs. maintenance costs
  - Business process reengineering vs. incremental architectural changes:
    - Revolution vs. evolution
  - Perfect architecture vs. time-to-market

Isolate Independent Concerns

- Data representation and communication protocols
- Security and communication
- Hide component technology:
  - Within an application, functionality (business model) is independent of CORBA/CCM, J2EE, .NET
- Hide middleware technology:
  - When two distant entities communicate, they should be ignorant of the technology used to transfer data between them

Examples

SNMP

- Protocol to transfer mgmt data between agents (managed entities) and managers (smart part of mgmt application)
- Monitoring of network devices
- No clear separation of concerns:
  - Application-level protocol
  - Data transfer protocol
  - Security
  - Representation of data in transit
  - Modeling of data

Event-Based Systems

- Heterogeneous event formats
- Heterogeneous middleware to exchange events
- My XSD is better than yours
- Translation can cause semantic loss
- Best practices call for:
  - Self-describing events
  - Ontologies
  - Separate data semantics aspect from communication aspect

Globus Toolkit

- Today's de facto standard middleware for Grids
- Good practices:
  - Components
  - Versioning
  - Auto-discovery of platform
  - Simple configuration: Makefile parameters are automatically propagated across code tree
  - Leverage well-known tools: GridFTP extends FTP
- Bad practices:
  - Reinvent the wheel: GNU software ignored
  - GridFTP 2.4 assumes TCP underneath. When SCTP appeared, GridFTP 2.4 would not work over SCTP. Required a complete reengineering → GridFTP 3.0.

Grid Applications (1/2)

- Good practices:
  - Avoid design by committee and over-engineering antipatterns
  - Code and test on real platforms
  - Create a user community and share experience
- Bad practices:
  - Reinvent the wheel:
    - GGF invented Grid services, began specifying OGSI, and then realized that Web services did 95% of what they needed
    - GGF has shadow WGs/RGs for many IETF, IRTF and DMTF WGs/RGs

Grid Applications (2/2)

- Bad practices (cont'd):
  - Code first, design afterward:
    - OGSA (arch) confused with OGSI (API)
    - OGSA/arch began several years after the other WGs
    - Scalability issues ignored until they showed up in practice (e.g., LDAP bottlenecks)
    - Impact of heterogeneity on Grid application design was underestimated
  - Poor skill matching:
    - Application domain experts cannot be turned into software engineering experts overnight
    - Expertise in MPP systems is not expertise in architecting worldwide Grid applications

More on Software Architecture

Architectural Views (1/3)

- Different views support different goals and uses
- Different people involved in a software project are interested in different views
- Clements et al.:
  - *Layered view:* tells you about your system's portability
  - *Deployment view:* lets you reason about your system's performance and reliability

Architectural Views (2/3)

- Kruchten:
  - *Logical view:*
    - Behavioral requirements (services that the system should provide)
  - *Process view:*
    - Concurrency, distribution, system integrity, fault tolerance
  - *Development view:*
    - Identification of software units that can be developed by different people/teams, cost evaluation, planning, reuse, portability, security
  - *Physical view:*
    - System's availability, reliability, performance, scalability

Architectural Views (3/3)

- Hofmeister et al.:
  - *Conceptual view:*
    - Functionality of system mapped to components and connectors
  - *Module view:*
    - Components and connectors mapped to subsystems and modules
  - *Execution view:*
    - Modules mapped to elements provided by runtime platform and hardware
    - Performance, recovery, concurrency, replication
  - *Code view:*
    - Deployment, versioning

Components and Connectors (1/2)

- Building blocks of the conceptual view
- A *component* is a group of objects that belong together:
  - Coarser grained than a Java bean
  - Examples:
    - Client
    - Server
    - Database
    - Layer in hierarchical system
  - To allow for composition, a component can be a subsystem of arbitrary size

Components and Connectors (2/2)

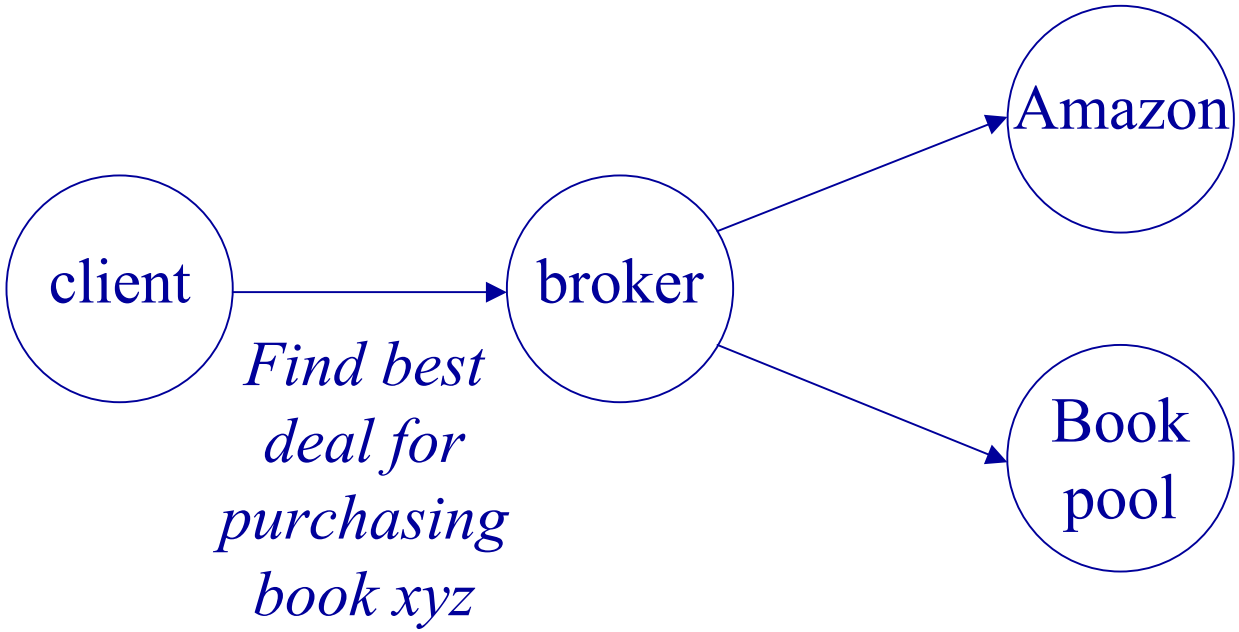
- A *connector* mediates interactions between components:
  - Two main functions:
    - Coordination
    - Data exchange
  - Examples:
    - RPC call
    - Client-server protocol
    - Database access protocol
    - LDAP
    - Asynchronous event multicast

Architectural Patterns

- Two books:
  - POSA1 by Buschmann et al.
  - POSA2 by Schmidt et al.

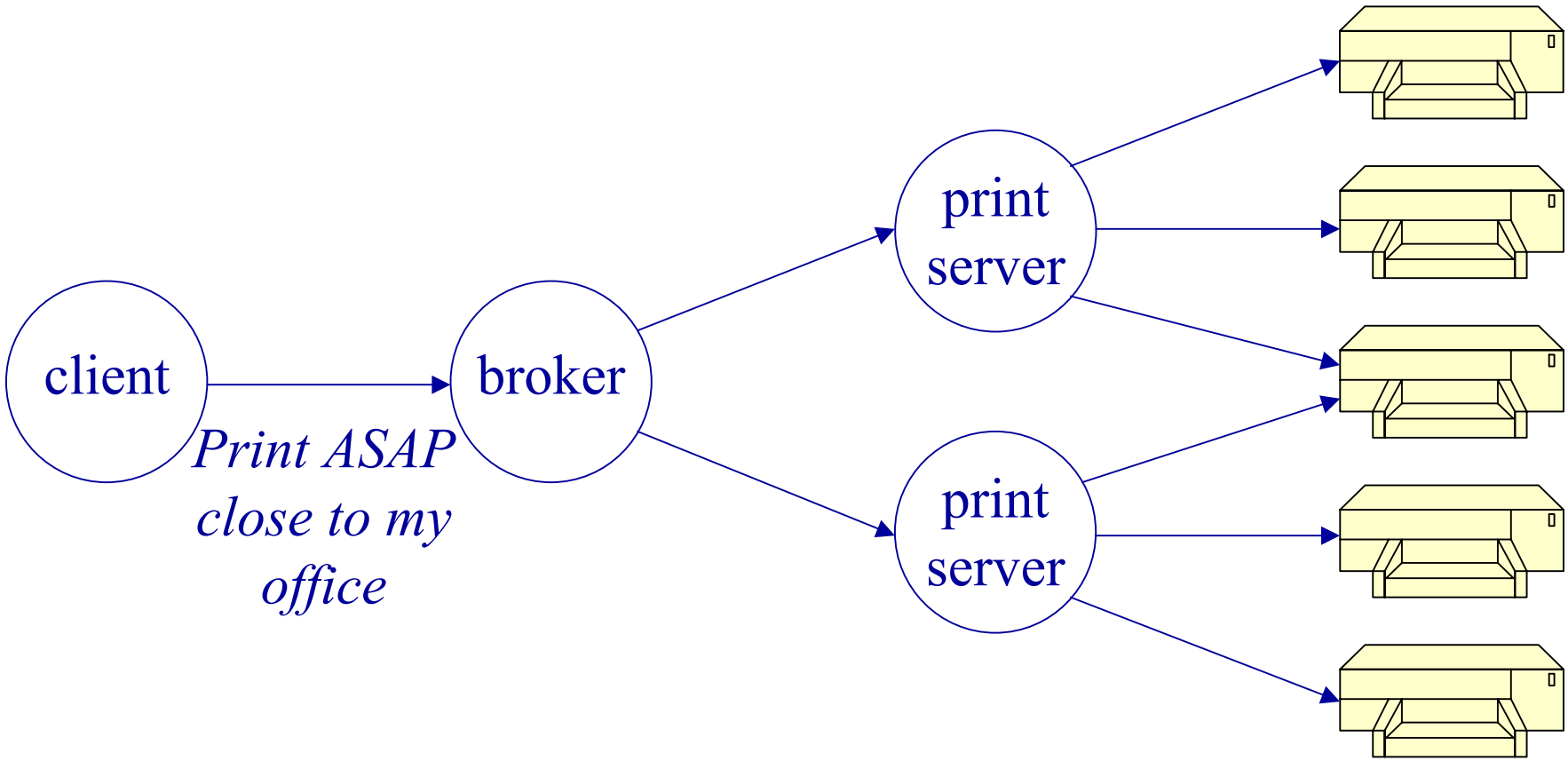
Broker (1/2)

- A useful pattern for service discovery :



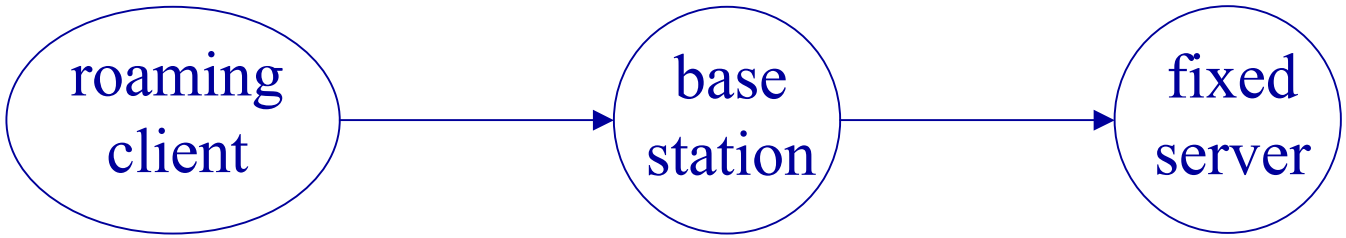
Broker (2/2)

- Several metrics:



Client-Dispatcher-Server

- Location transparency in nomadic computing:



Architectural Styles

- Old name for architectural patterns
- Objective: build catalogs of alternatives for solving a given architectural problem
- Can be mapped to components+connectors
- Clements et al.:
  - Pipes and Filters
  - Shared Data (database, knowledge base)
  - Publish-Subscribe
  - Client-Server
  - Peer-to-Peer
  - Communicating Processes

Research Perspectives

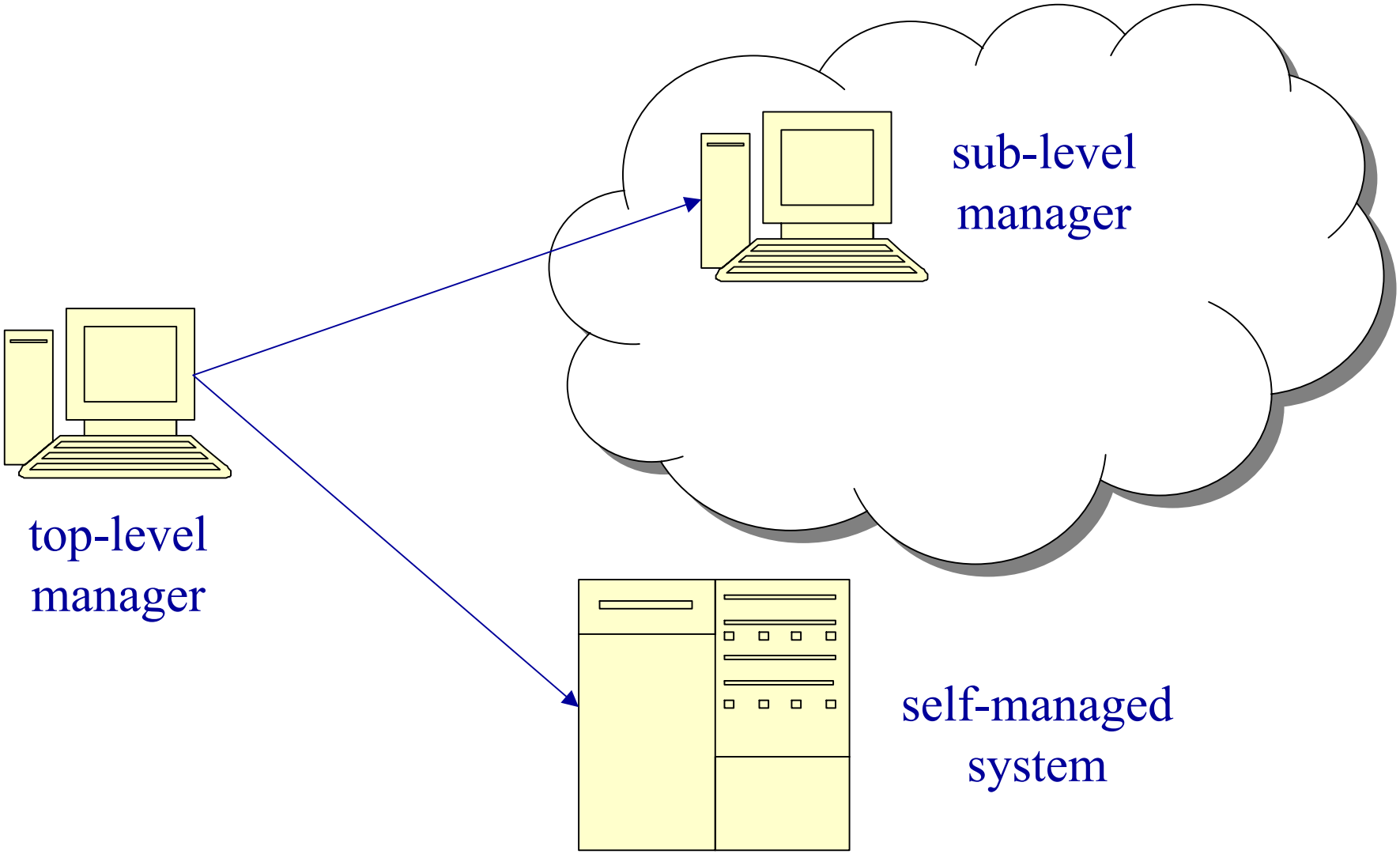
Automation: One Step Beyond (1/2)

- Software development automation: from models to code
- Step 1: automate coding:
  - From assembling low-level design patterns ("mental building blocks") to assembling pieces of code
- Step 2: automate low-level design:
  - From assembling high-level design patterns to assembling low-level design patterns

Automation: One Step Beyond (2/2)

- Step 2 (cont'd):
  - Architecture Description Languages:
    - Medvidovic and Taylor, *IEEE TSE*, Jan 2000
  - Model-Driven Architecture:
    - Platform-Independent Model (business model)
    - Platform-Specific Model
    - Automate transformation PIM → PSM
  - UML 2.0
  - Aspect-oriented programming
  - Generative programming:
    - Ultimate for service discovery

Self-Managed Systems



SOA for Integrated Management

- IM = integrate management of networks, systems, applications and services + customer care/helpdesk
- SOA-IM = evolution of WIMA
- From tight coupling (OIDs in SNMP) to loose coupling (Web Services)
- Service discovery in very heterogeneous and changing environments:
  - Platform independence
  - Middleware independence
  - Component independence
  - Data model independence
  - Protocol independence

Architecture of Grid Software

- Analyze existing Grid applications
- Identify and document architectural issues:
  - Especially scalability issues
- Produce a set of architectural antipatterns for Grids
- Produce a catalog of architectural patterns to help software engineers design Grid applications
- Input to GGF OGSA WG

Further Reading

- <http://www.sei.cmu.edu/architecture/bibliography.html>

- Technical aspects:
  - L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, 2nd Edition, Addison-Wesley, 2003.
  - M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.
  - P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R.Little, R. Nord and J. Stafford, *Documenting Software Architectures: Views and Beyond*, Addison-Wesley, 2002.
  - C. Hofmeister, R. Nord and D. Soni, *Applied Software Architecture*, Addison-Wesley, 2000.
- Business aspects:
  - L. Hohmann, *Beyond Software Architecture: Creating and Sustaining Winning Solutions*, Addison-Wesley, 2003.