Introduction to Software Architecture and Design

Cesare Pautasso

http://www.pautasso.info

Contents

- Architecture 101
- When do you need an Architect?
- Why do we need Software Architecture?
- Architecture within the Software Development Process
- Course Overview

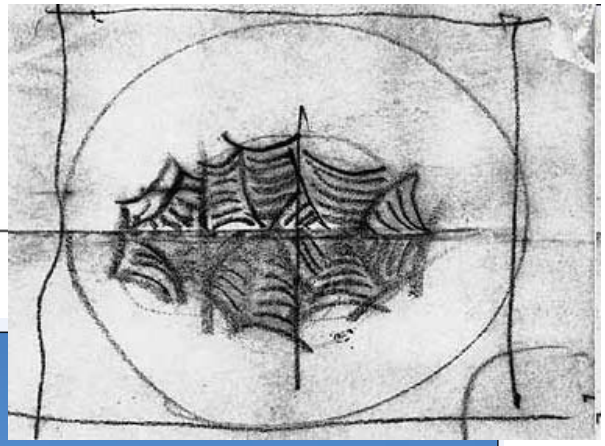

Software Architecture

De Architectura

firmitatis utilitatis venustatis

- **Durability:**
  the building should last for a long time without falling down on the people inside it
- **Utility:**
  the building should be useful for the people living in it
- **Beauty:**
  the building should look good and raise the spirits of its inhabitants

Vitruvio, *De Architectura*, 23BC



Architecture

The Art and Science of Building

- Architects are not concerned with the creation of building technologies and materials—making glass with better thermal qualities or stronger concrete.
- Architects are concerned with how building materials can be *put together* in desirable ways to achieve a **building suited to its purpose.**
- The **design process** that *assembles* the parts into a useful, pleasing (hopefully) and cost-effective (sometime) whole that is called “architecture”.

Università della Svizzera italiana

Facoltà di scienze informatiche

Architecting, the planning and building of structures, is as old as human societies – and as modern as the exploration of the solar system

Eberhardt Rechtin
1991

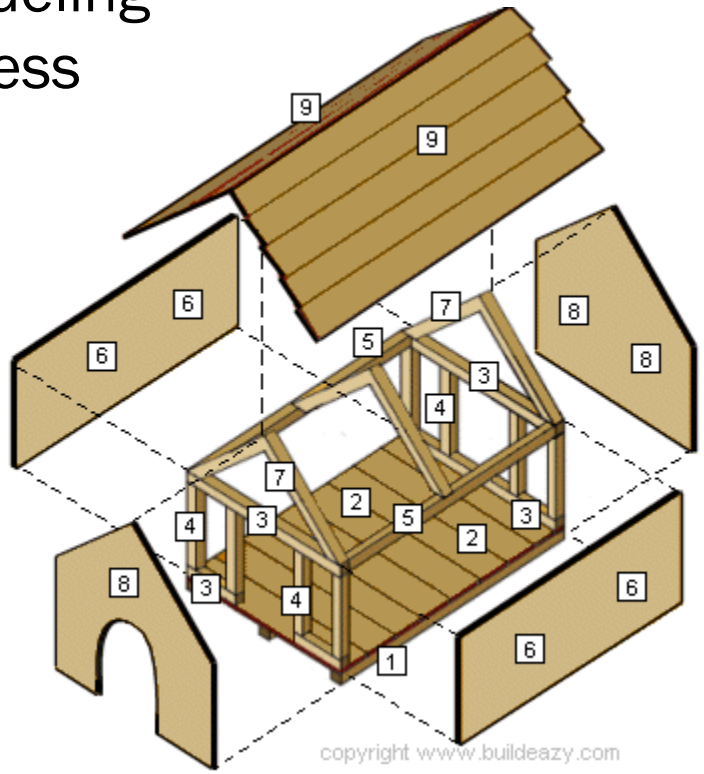
17.2.2009


Università della Svizzera italiana
Facoltà di scienze informatiche
When do you need an architect?
It depends what you are trying to build...

Small

One person can do it alone with:

- Minimal modeling
- Simple process
- Simple tools



Medium

Better to build a house as a team using some:

- Modeling plans
- Well-defined process
- Power tools

Large

You definitely need an architect for planning a skyscraper!

How large?

- Always choose the tools and design strategy appropriate for the **size** of the project you're working on

How large?

- How much **time to build** it?
- **How many people** in the project?
- How much did it **cost**?

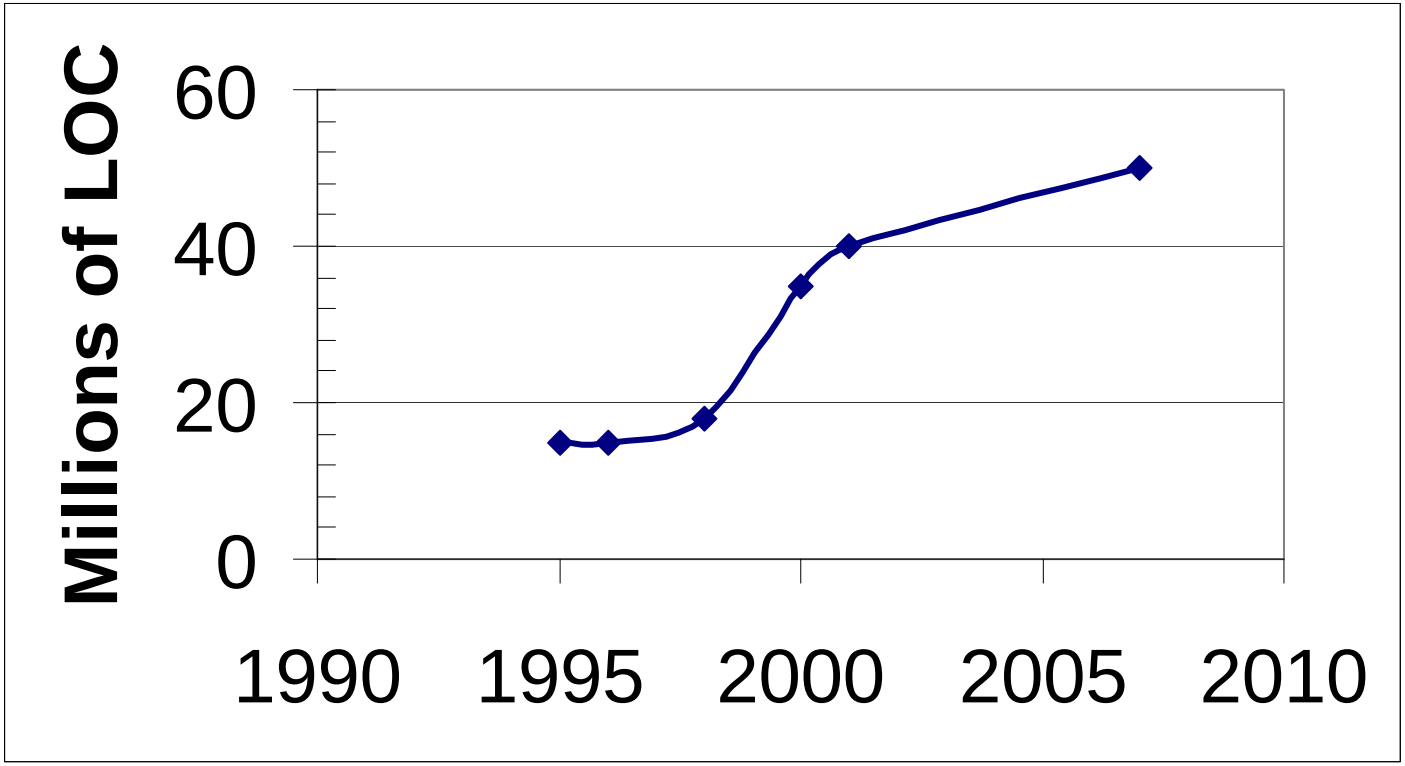
- Hard to measure software size:
  - Estimate the “lines of code” (LOC)
  - Measure the deployment size (GB)

How large?

Name	Size	Time
Indusha	4-5 People (Java)	1 semester
Luca	2 People (C++, 4k LOC)	2 months
Athos	4 People (C, Hardware)	1 year
Alan	4 People (PHP, ~100 “pages”)	3 years
Andrea	1 Person (Bachelor Prj, PHP 40-50 files)	2.5 months
Alessio	15-20 People (Wiki, Ruby, Atelier Project, 50 Ruby Files)	1 semester
Labinot	5 People (C#, 70-80 classes)	8 months
Alessio G.	1 Person (Tapestry, Java5, 50 classes)	6 months
Danilo A.	Robots (C++/Java/C, 30 big classes)	6 months

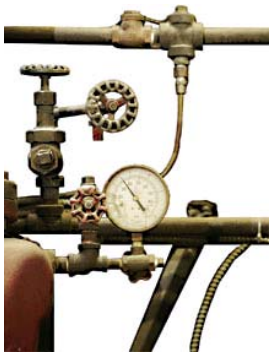
Large Software Project

- Lines of code: 50 Million
- Number of developers: 2000
- Time to compile: 24 hours
- Time to ship: 5 years (from previous release)



Software Architecture

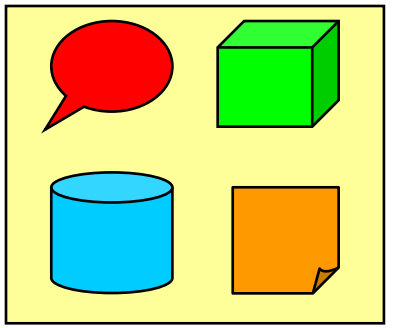
As the size and complexity of a software system increase, the **design decisions** and the **global structure** of a system become more important than the selection of specific *algorithms* and *data structures*.



Context

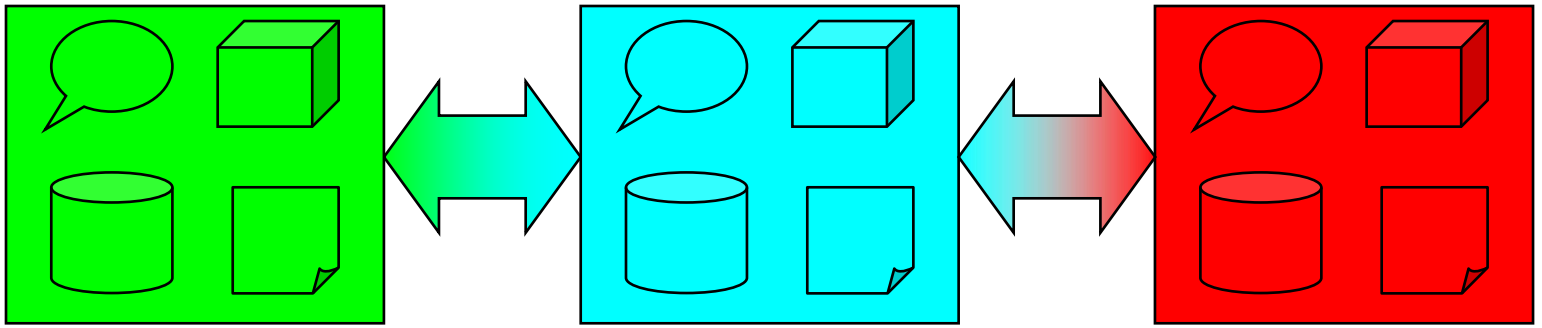
- Software Engineering
- Programming Languages
- Algorithms and Complexity
- Databases

How to build components from scratch



- Software Architecture
- Component-based Software Engineering

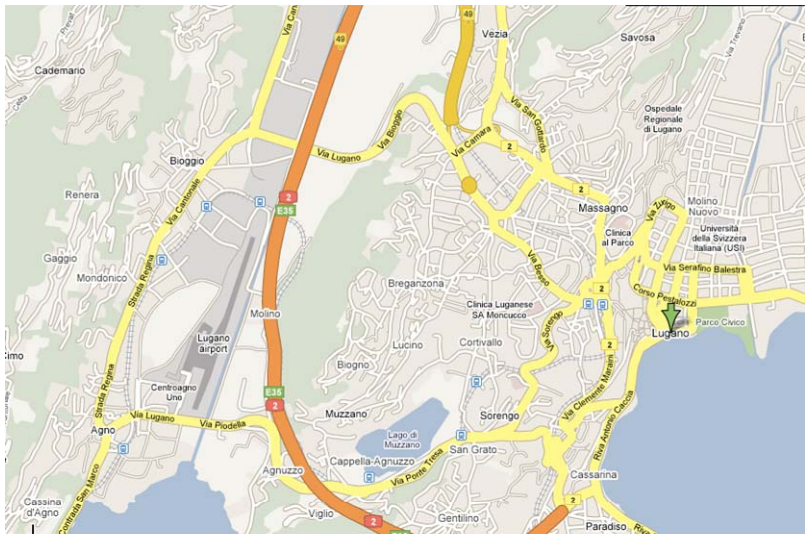
How to design large systems out of reusable components



Why SW Architecture?

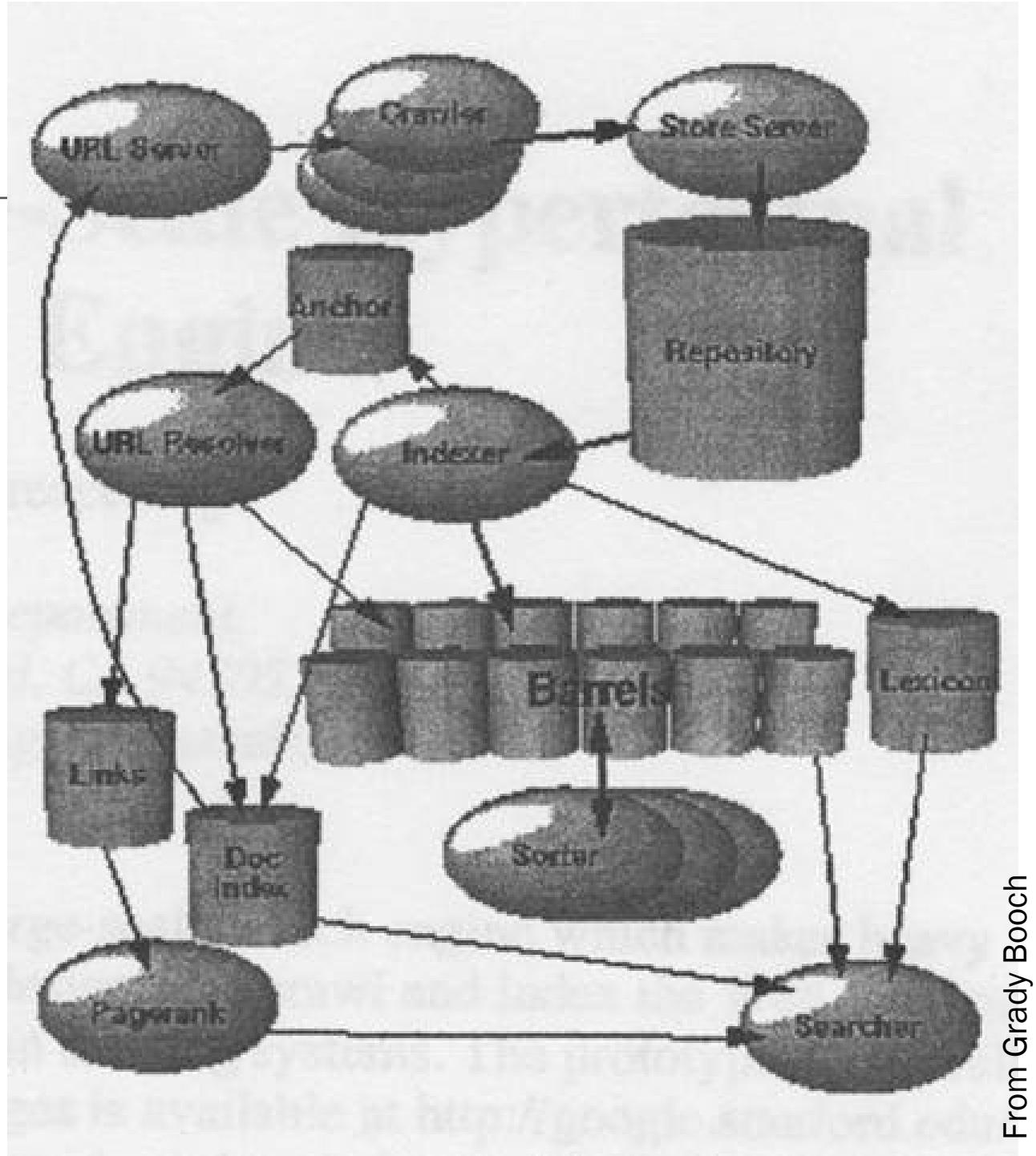
1. Manage complexity of large software projects through abstraction
2. Communicate, remember and share global design decisions among the team
3. Visualize and represent relevant aspects (structure, behavior, deployment, …) of a software system
4. Understand, predict and control how the design impacts quality attributes of a system
5. Define a flexible foundation for the maintenance and future evolution of the system

Abstraction



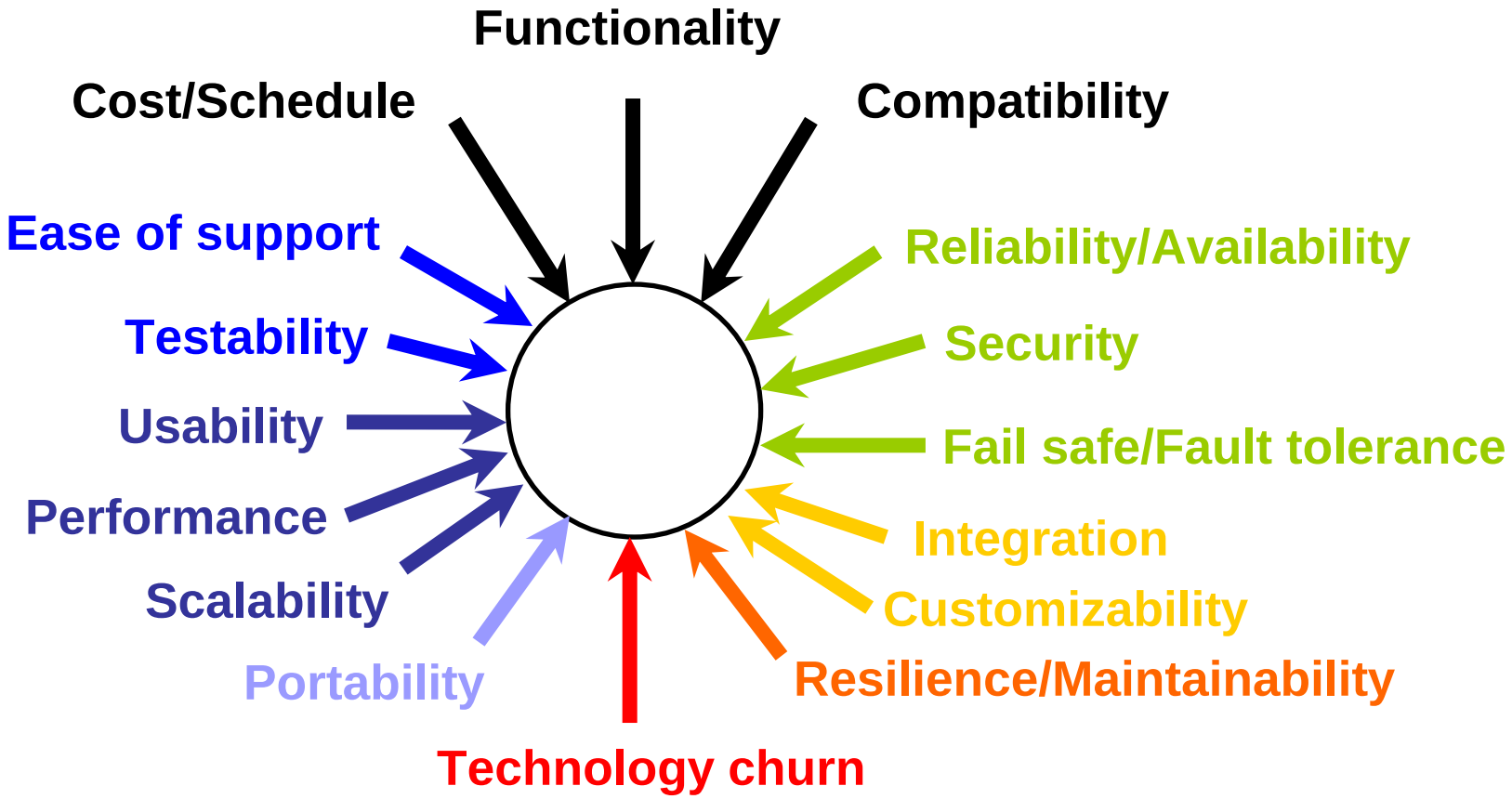
- The hard part is to know which details to leave out and which should be emphasized

Representation



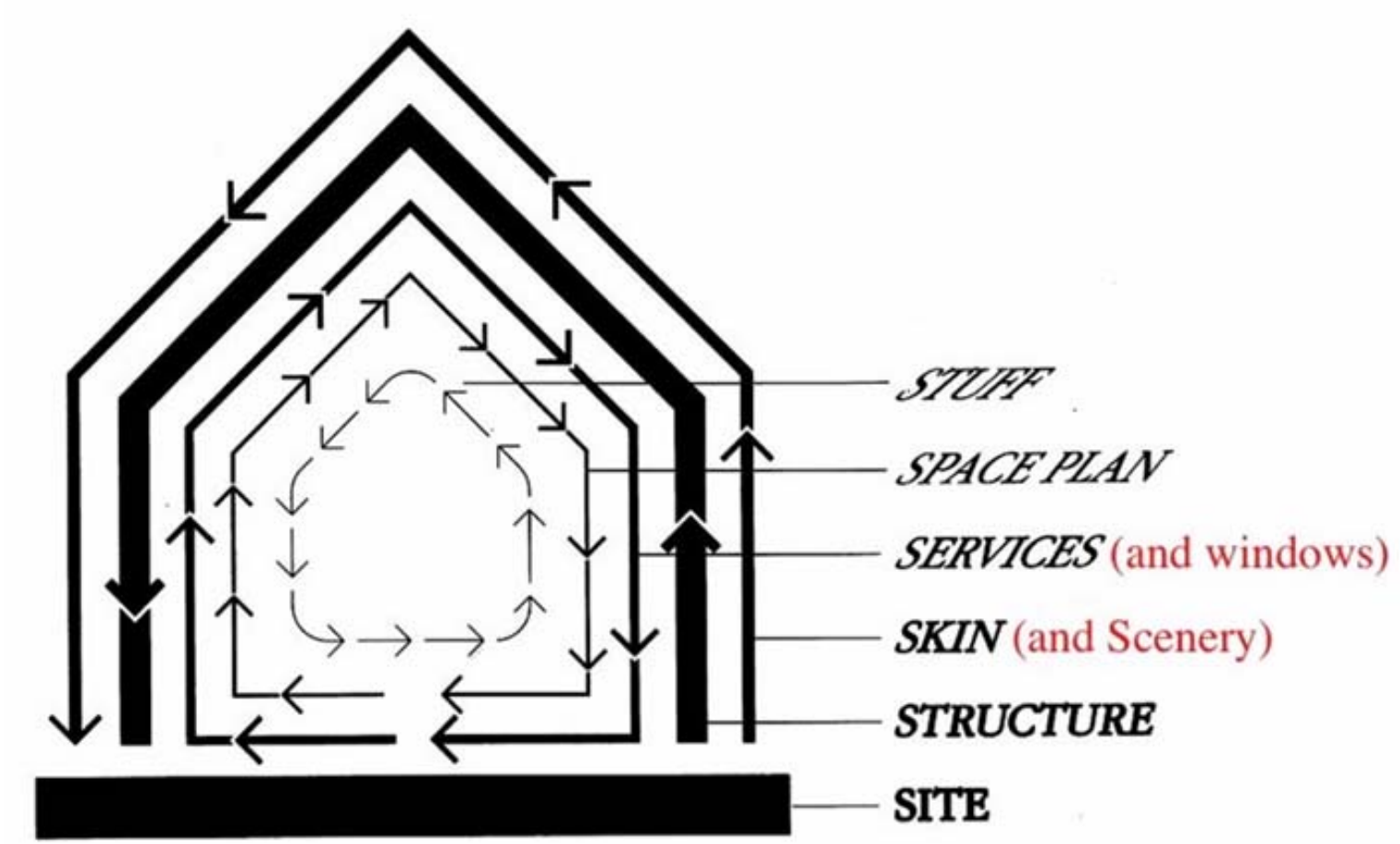
From Grady Booch

Quality Attributes



Adapted From Grady Booch

Future Evolution



Shearing layers of buildings:
RATES OF COMPONENT CHANGE

From Stewart Brands

Course Overview

Software Architecture

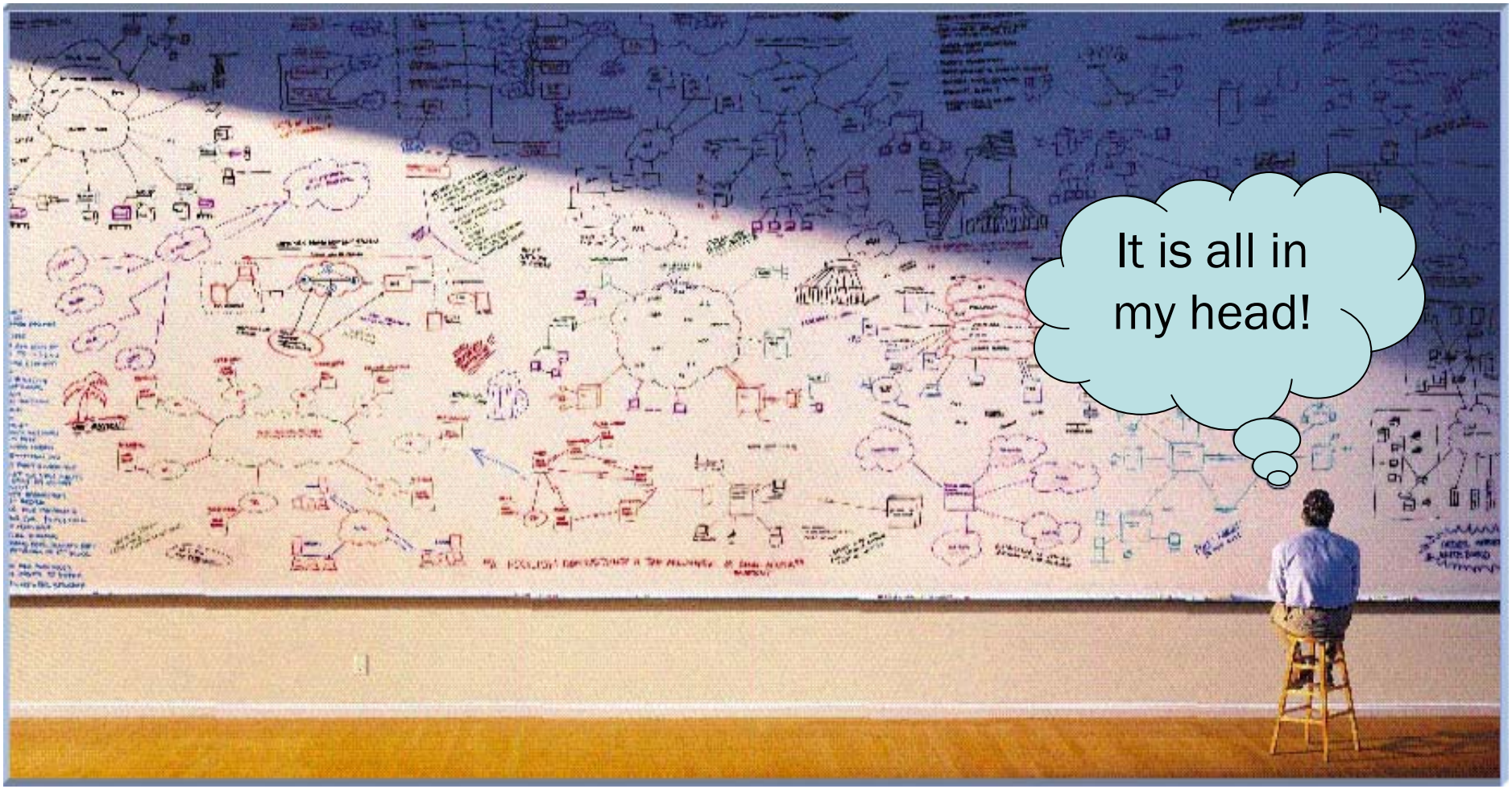
- Defining Software Architecture: Basic Concepts
- System Decomposition vs. Software Composition
- Architectural Views: Logical, Physical, Process, Development;
- Design Principles: Simplicity, Abstraction, Separation of Concerns, Encapsulation, Information Hiding
- Patterns and Anti-Patterns: Avoiding Common Design Mistakes
- Component Models
- Composition Techniques
- Architectural Styles

Lab

- Modeling Architectures: Architectural Description Languages
- Visualization of Architectures
- Evaluating Quality Attributes and Non-Functional Properties
- Architectural Decision Modeling
- API Design Techniques

Subject to change

Software Architecture



References

- Eberhardt Rechtin, **Systems Architecting: Creating and Building Complex Systems**, Prentice Hall 1991
- Henry Petroski, **Small Things Considered: Why There Is No Perfect Design**, Vintage, 2004
- Ian Gordon, **Essential Software Architecture**, Springer 2004
- Christopher Alexander, **The Timeless Way of Building**, Oxford University Press 1979
- Stewart Brand, **How Buildings Learn**, Penguin 1987
- Grady Booch, **Handbook of Software Architecture**, http://booch.com/architecture/

Università della Svizzera italiana

Facoltà di scienze informatiche

Defining Software Architecture

Prof. Cesare Pautasso

http://www.pautasso.info

Contents

- The role of the Software Architect
- Architecture and the Software Development Cycle
- Defining Software Architecture
  - Prescriptive vs. Descriptive Architecture
  - Architectural Degradation: Drift and Erosion
  - M-Architecture vs. T-Architecture
  - Solution vs. Product

Who is a software architect?

- William (Bill) H. Gates,
  Chief **Software Architect**, Microsoft

(until June 2006)

Università della Svizzera italiana | Facoltà di scienze informatiche

IT Architect Job Ad

Wir bieten:

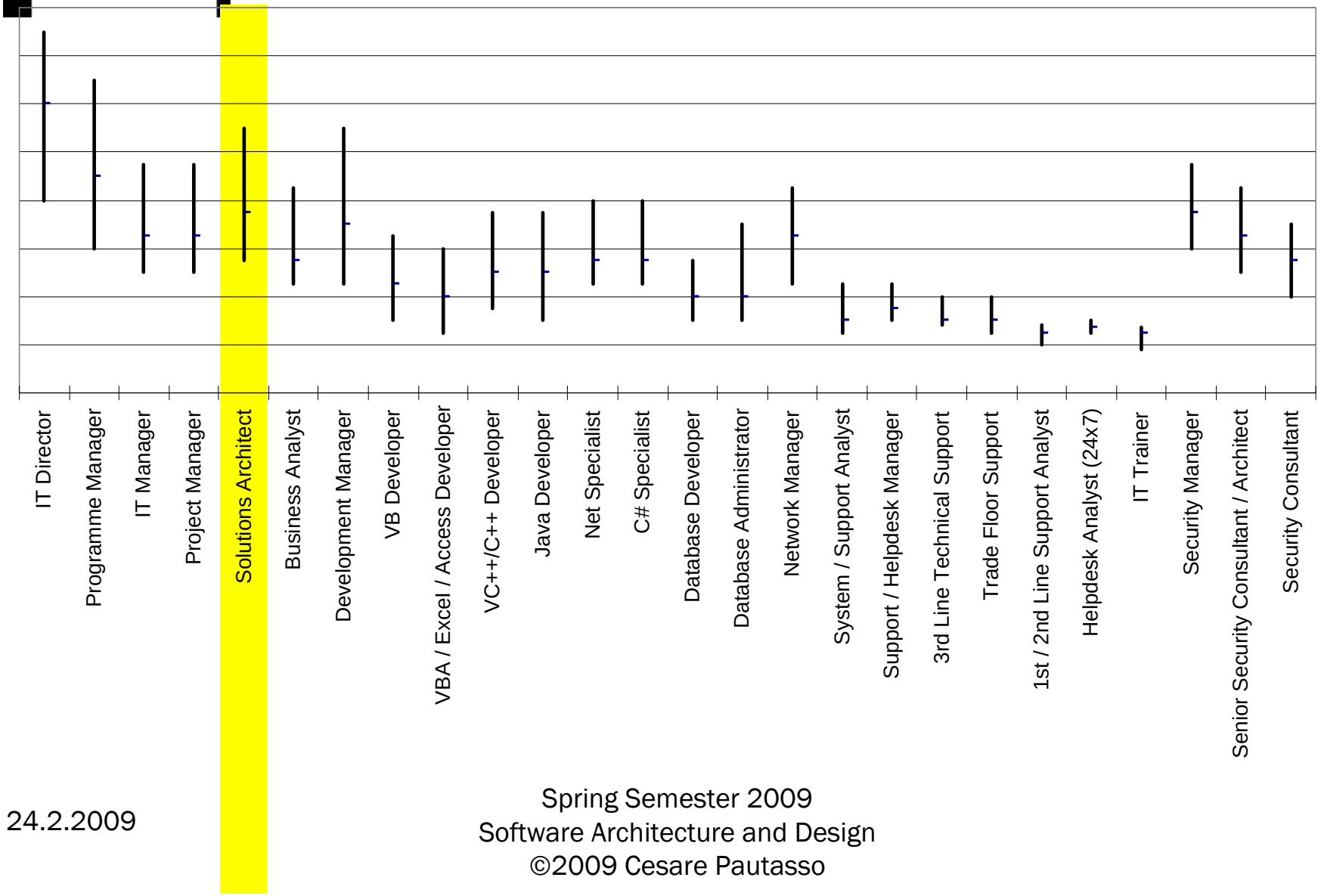
- Mitarbeit in der Definition der Targets für die NextGen Architecture Platform für IT Core Banking Solutions & Services
- Unterstützung bei der Planung und der Erstellung von Dokumentationen, Leitfaden und Konzeptbeschreibungen für die definierten IT & Platform Architecture Standards
- Unterstützung in der Kommunikation der IT & Platform Architecture Standards an Interessenten
- Koordination der Projektarbeit mit Projekt Mitarbeitern
- Unterstützung des Head of Architecture Officer in den Koordinations- und Kontrollaufgaben betreffend der Umsetzung der IT & Platform Architecture Standards sowie in den ScoreCard Rapportierungen
- Schnittstellenfunktion zwischen den verschiedenen Architekturschnittstellen und Interessenten
- Enge Zusammenarbeit mit den entsprechenden Stakeholdern inklusive projektrelevanten Gremien

Sie bieten

- Fach- oder Hochschulabschluss (Wirtschaftsinformatik/Informatik)
- **Mind. 5 Jahre Erfahrung** in der IT Architektur (Applikation- und Technologie-Architektur) und der Software-Entwicklung in einem Grossunternehmen im Finanzdienstleistungsbereich
- Fundierte Kenntnisse im Host-Bereich und in den entsprechenden Technologien wie z.Bsp. PL1, J2E, SOA (CORBA, WebServices) und High-Volume Transaktionssysteme sowie in der Datenmodellierung und im Reengineering
- Verhandlungssichere Kommunikation in Deutsch und Englisch

Posted 18.2.2008, CS Zurich

Value of the Architect



From Michael Page Intl

Università della Svizzera italiana

Facoltà di scienze informatiche

Customers

Marketing

Analyst

IT Operations Manager

Architect

Application Support Helpdesk

Deployment

Tester - QA

Modeler

Developer

Product Manager

Model the problem

Define Requirements

Architect & Design

Program Code

Test

Deploy

Manage

Measure Performance

Manage Change

Operations

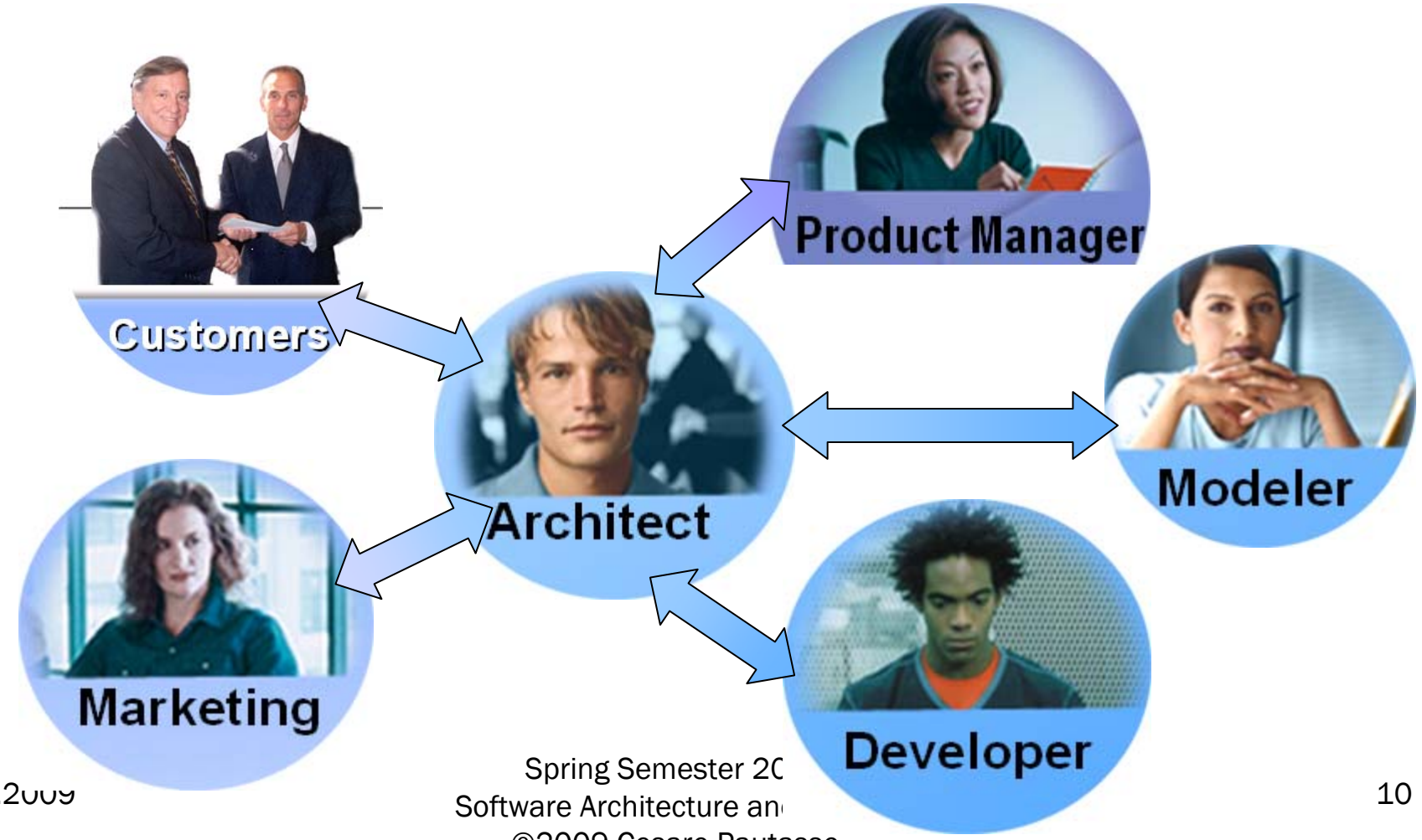
Development

Role of the Architect

1. Software Engineering Lead
   - Excellent software engineering skills
   - Lead technical development team by example
   - Solve the hard problems
   - Understand impact of decisions
   - Defend architectural design decisions
   - Promote good development practices
   - Plan and manage software releases

Role of the architect

2. Facilitate Communication



Role of the architect

3. Technology Expert
   - Know and understand relevant technology
   - Evaluate and influence the choice of 3rd party frameworks, component and platforms
   - Track technology development
   - **Know what you don't know**

Role of the architect

4. Risk Management
   - Estimate and evaluate risks associated with design options and choices
   - Document and manage risks, making the whole team (incl. management) aware
   - Prevent disasters from happening

Architect Soft Skills

Technical
- Creative problem solver
- Practical/pragmatic
- Insightful, Investigative
- Tolerant of ambiguity, willing to backtrack, seek multiple solutions
- Good at working at an abstract level

Consulting
- Committed to others' success
- Empathetic, approachable
- An effective change agent, process savvy
- A good mentor, teacher
- Visionary
- Entrepreneurial

Leadership
- Charismatic and credible
- Committed, dedicated, passionate
- Know how to motivate teams

Organizational politics
- Able to see from and sell to multiple viewpoints
- Confident and articulate
- Ambitious and driven
- Patient and not
- Resilient
- Sensitive to where the power is and how it flows in an organization

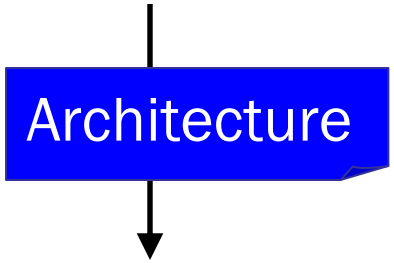
Adapted from Bredemeyer.com

Software Architecture

and the Software Development Lifecycle

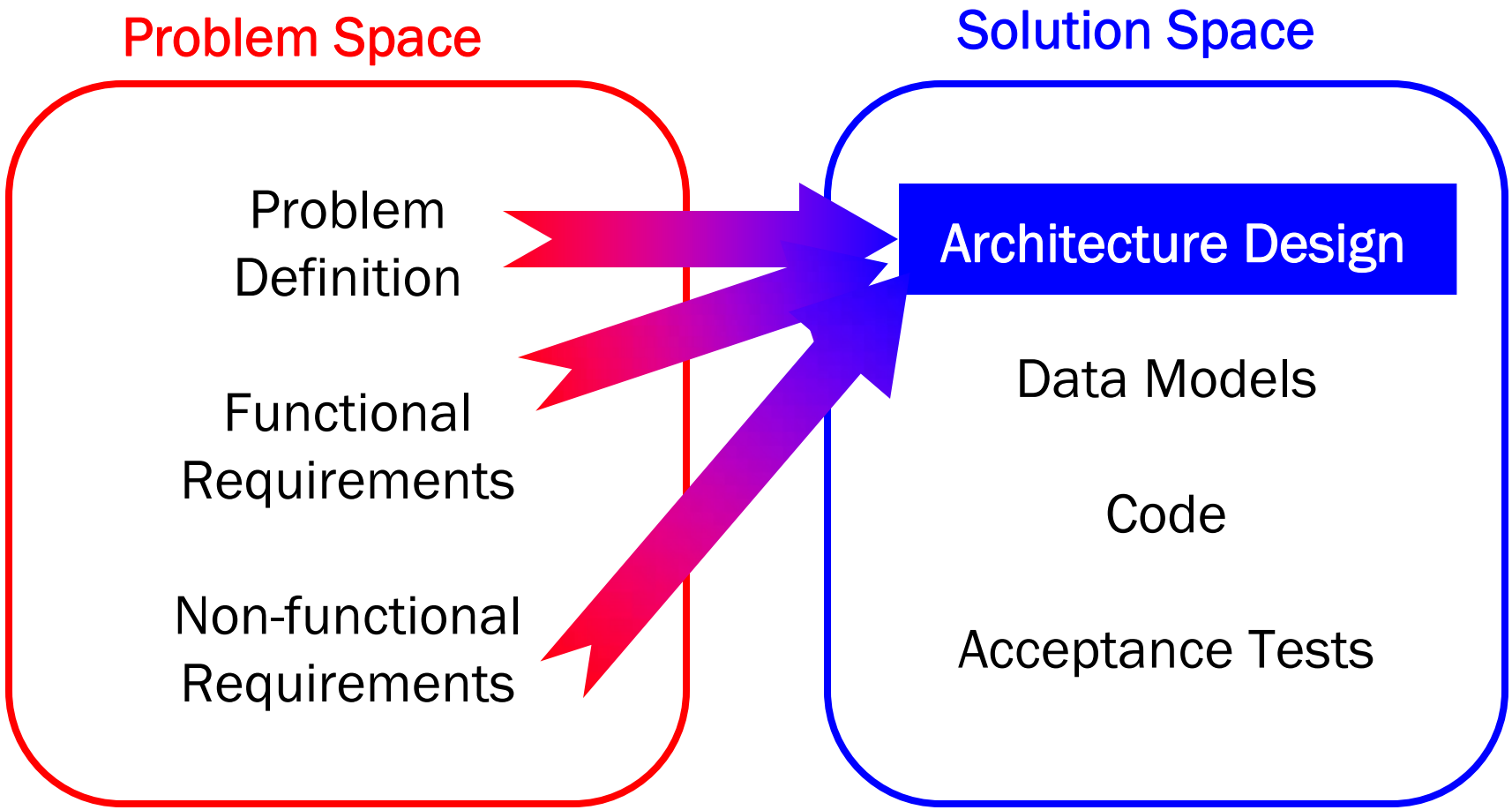
Waterfall Model

1. Requirement Analysis
2. Design

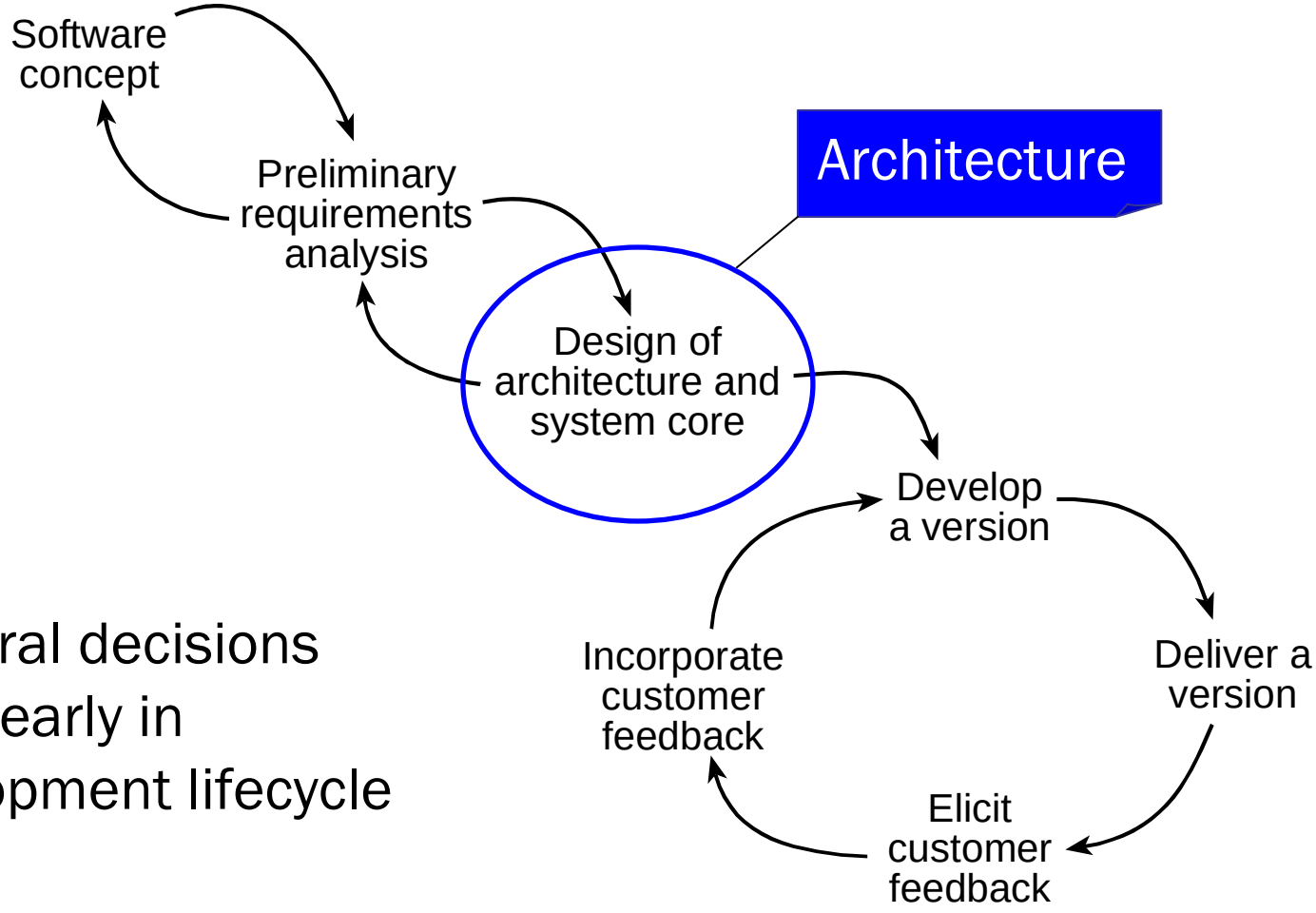


3. Implementation
4. Testing
5. Maintenance

Bridge the Gap



Evolutionary Model

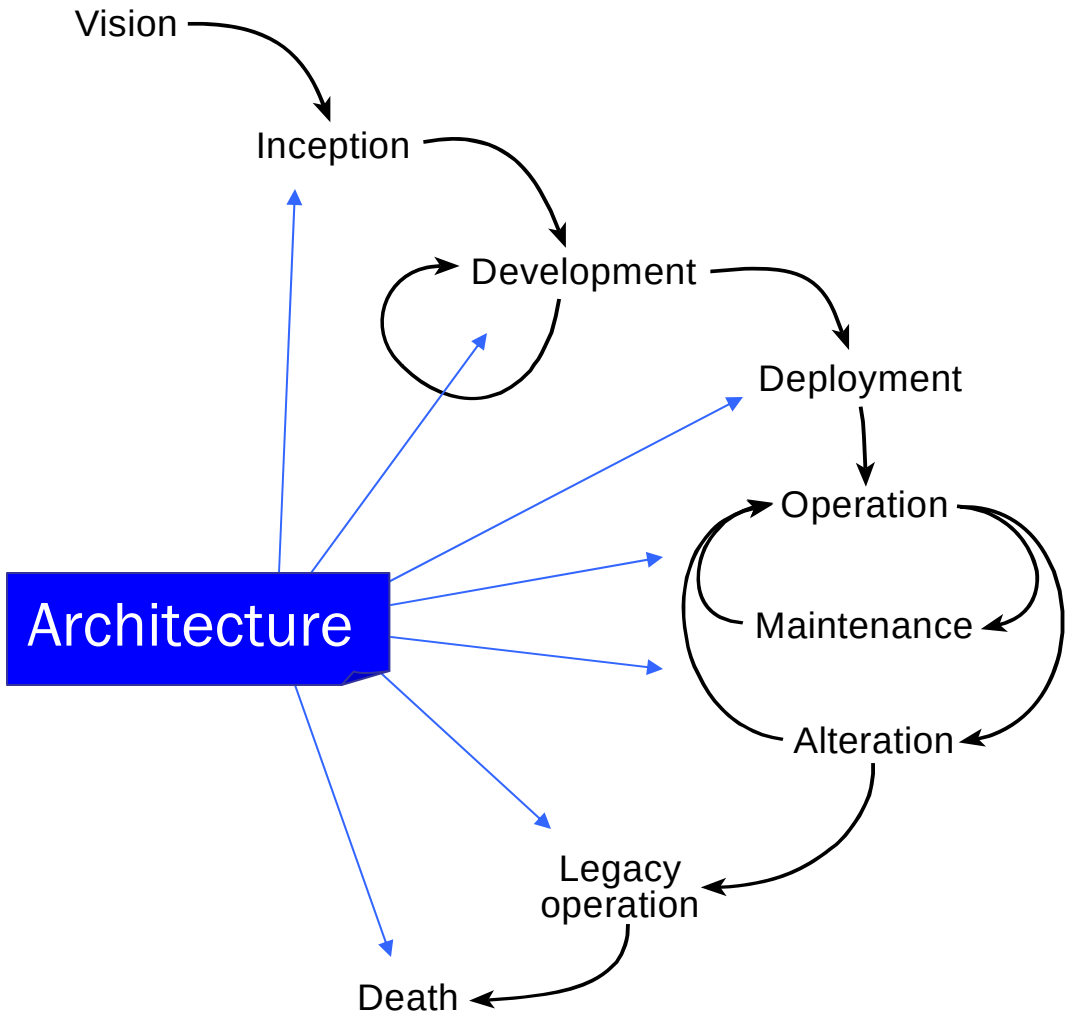


Architectural decisions are made early in the development lifecycle

From John Reekie, Rohan McAdam

System Lifecycle

Architectural decisions affect the whole lifetime of a system



From John Reekie, Rohan McAdam

Defining Software Architecture

Architecture is what architects do

Basic Definition

- A software system's architecture is the set of **principal design decisions** made about the system.

Architecture = {Principal Design Decision}

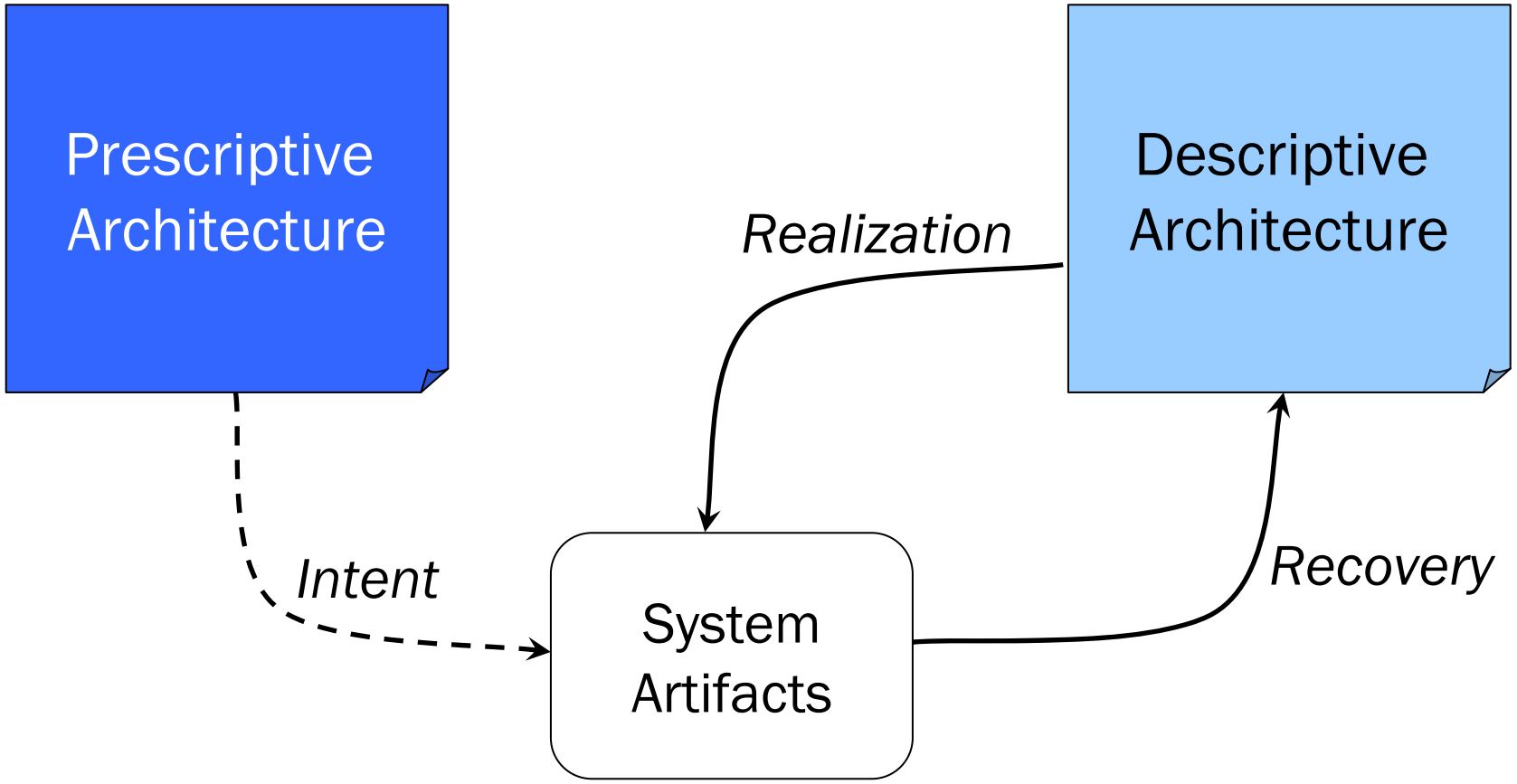
- It is the blueprint for a software system's construction and evolution

From N. Taylor et al

Design Decisions

- Design decisions cover all aspects of the system:
  - Structure
  - Behavior
  - Interaction
  - Deployment
  - Non-functional properties
  - Implementation
- **Principal** Design Decisions:
  - the "important ones" depending on the goals of the stakeholders
- Decisions are made over time
- Decisions are changed over time
- Decisions depend on other decisions

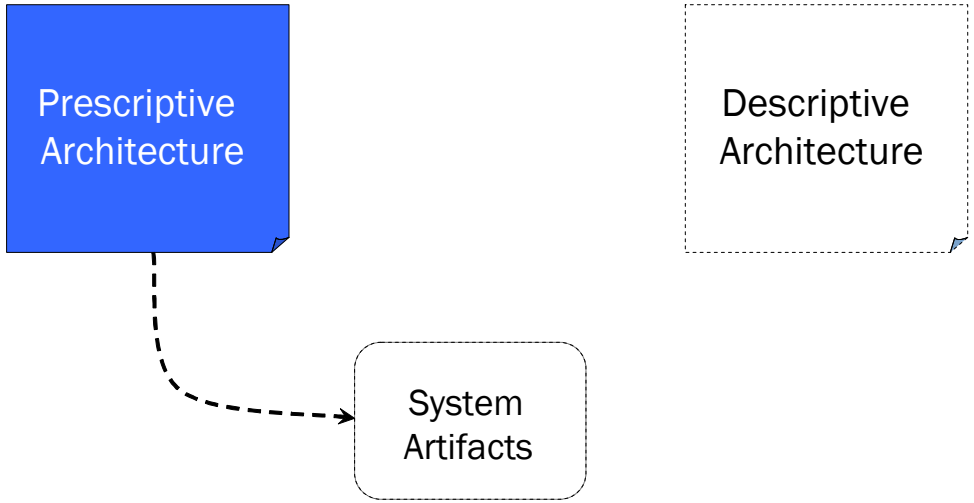
Prescriptive vs. Descriptive



From N. Taylor et al

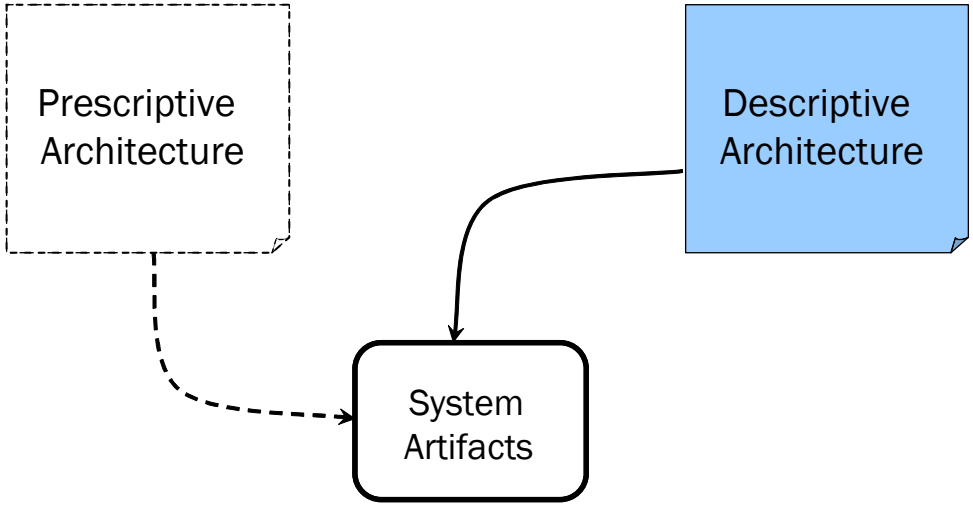
Green Field Development

- At the beginning, systems designed and implemented from scratch only have a prescriptive architecture

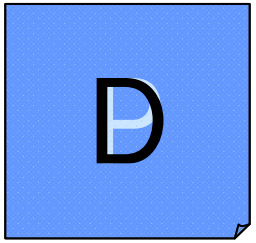


Brown Field Development

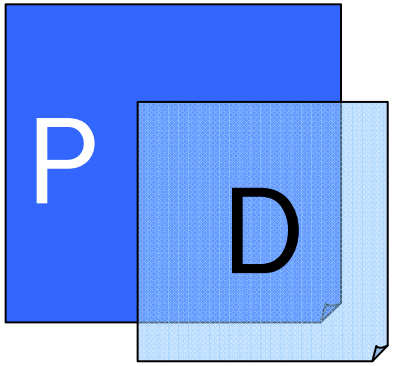
- Systems developed reusing existing components already have a descriptive architecture
  (and may have an empty prescriptive architecture)



Architectural Degradation



Ideal Case

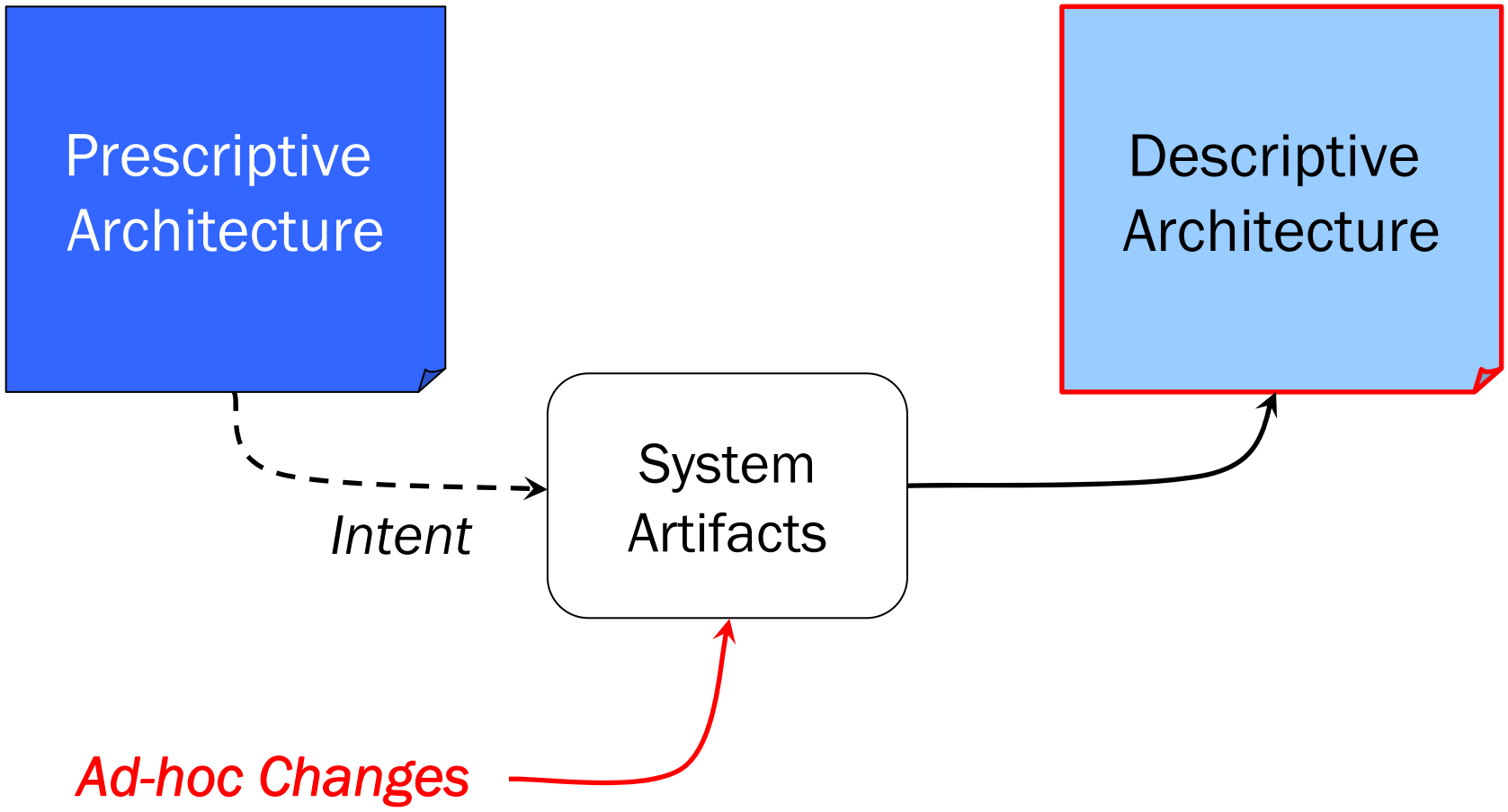


Realistic Case

- Ideal Case (P = D)
- D always a perfect realization of P

- Not all P decisions can be implemented
- Over time, P and D change independently and drift apart

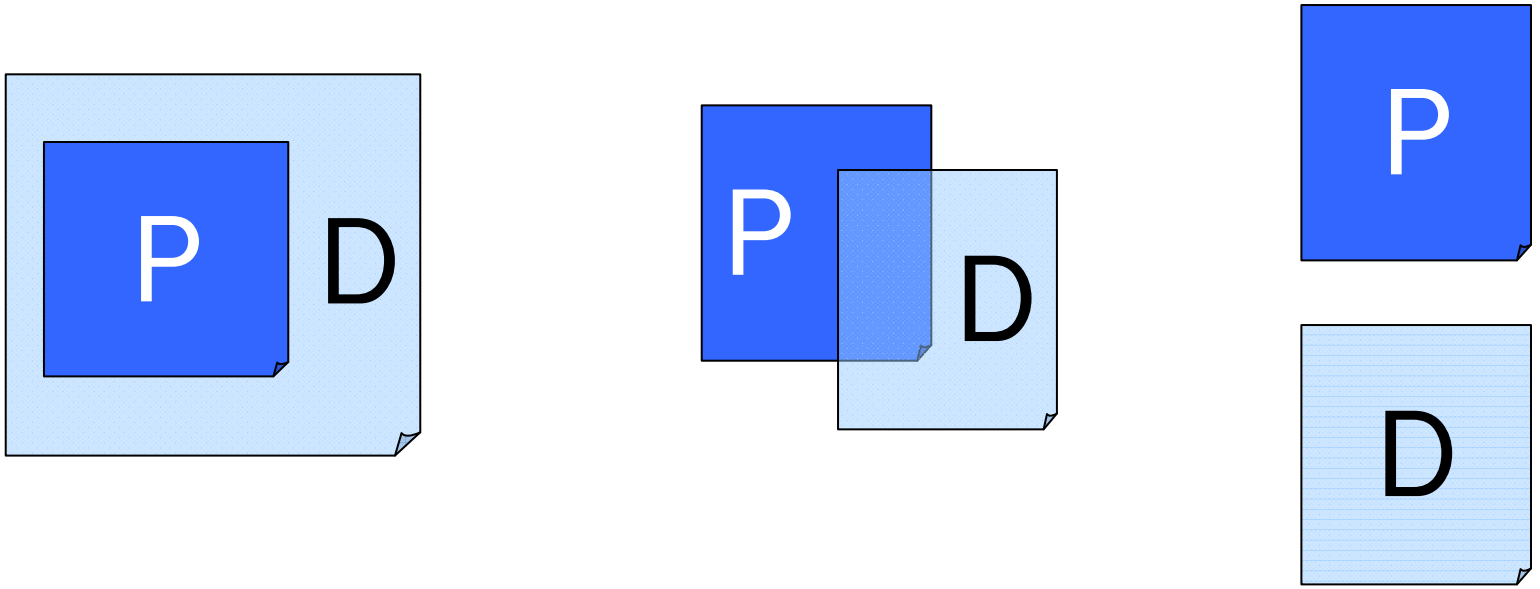
Causes of Architectural Drift



From N. Taylor et al

From Drift to Erosion

- Over time, decisions are added to D



- **Drift**: new D decisions do not violate P
- **Erosion**: D decisions violate P decisions

From N. Taylor et al

Solution vs. Product

- A *solution architect* solves the problems of **one customer** by designing a solution architecture made of **multiple products** that are integrated and reused as components
- A *product architect* designs **one product** architecture that can solve the problems and satisfy all the requirements of **multiple customers**

From O. Zimmermann

M-architecture, T-architecture

- **Principal** Design Decisions:
  - the "important ones" depending on the goals of the **stakeholders**

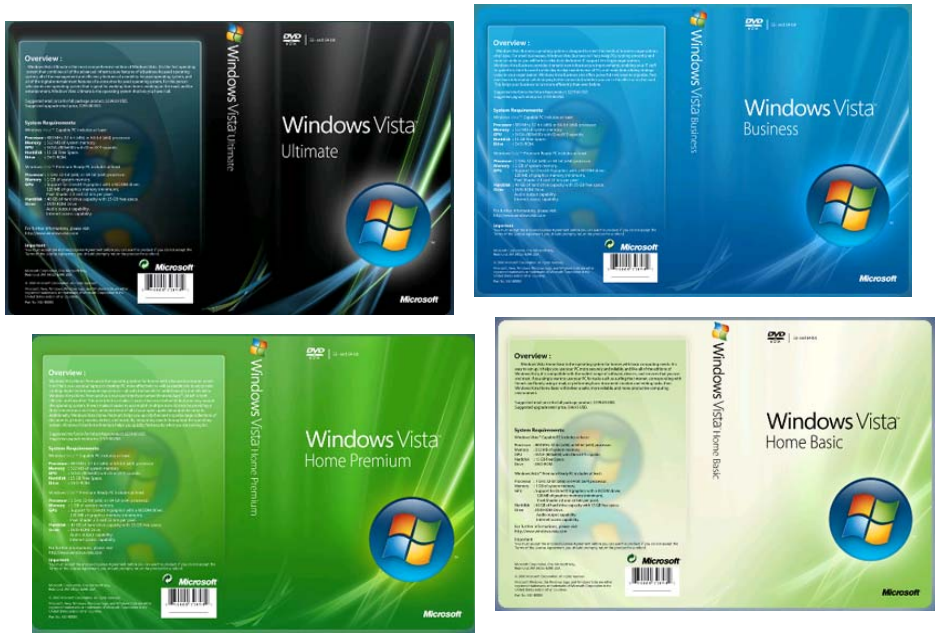
- Marketing Architecture
- Describe how to market and sell (business model, licensing model) the system to customers

- Technical Architecture
- Prescribe how to build deploy and configure the system (styles, patterns, components, connectors)

From L. Hohmann

The $10'000 Boolean Flag

- Marketing Architecture
- 4 Different Products:



- Technical Architecture
- For multiple "products" there can be a single technical architecture and a single codebase.
- Simple configuration flags can be used to switch between the different M-architectures perceived by customers

Thursday Exercise

SI 006 – 8:30-10:30

References

- Richard N. Taylor, Nenad Medvidovic, Eric M. Dashofy, **Software Architecture: Foundations, Theory and Practice**, John-Wiley, January 2009
- Luke Hohmann, **Beyond Software Architecture: Creating and Sustaining Winning Solutions**, Addison-Wesley, February 2003
- Ian Gorton, **Essential Software Architecture**, Springer, April 2006
- Grady Booch, **Handbook of Software Architecture**, http://booch.com/architecture/

Università della Svizzera italiana

Facoltà di scienze informatiche

Software Components

Prof. Cesare Pautasso

http://www.pautasso.info

Università della Svizzera italiana

Facoltà di scienze informatiche

Contents

- Components
  - Defining Components
  - Classifying Components
  - Component Frameworks
  - The Problem of Heterogeneity and Interoperability

Architecture

Purpose

Customer Requirements

Quality

Stakeholders

Design Plan (Decisions)

Map:

Function → **Structure**

Behavior

Abstraction

Viewpoint

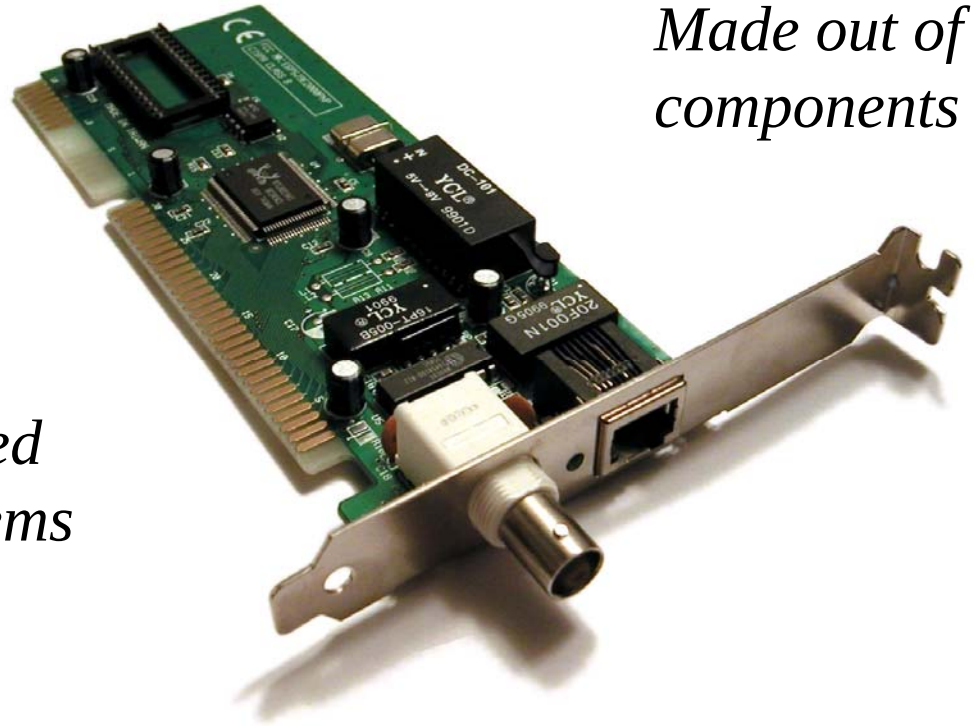
Interfaces
- Mediation

Components
- Processing
- Data/State

Connectors
- Communication
- Composition
- Collaboration

Hardware Component

- Reusable unit of composition

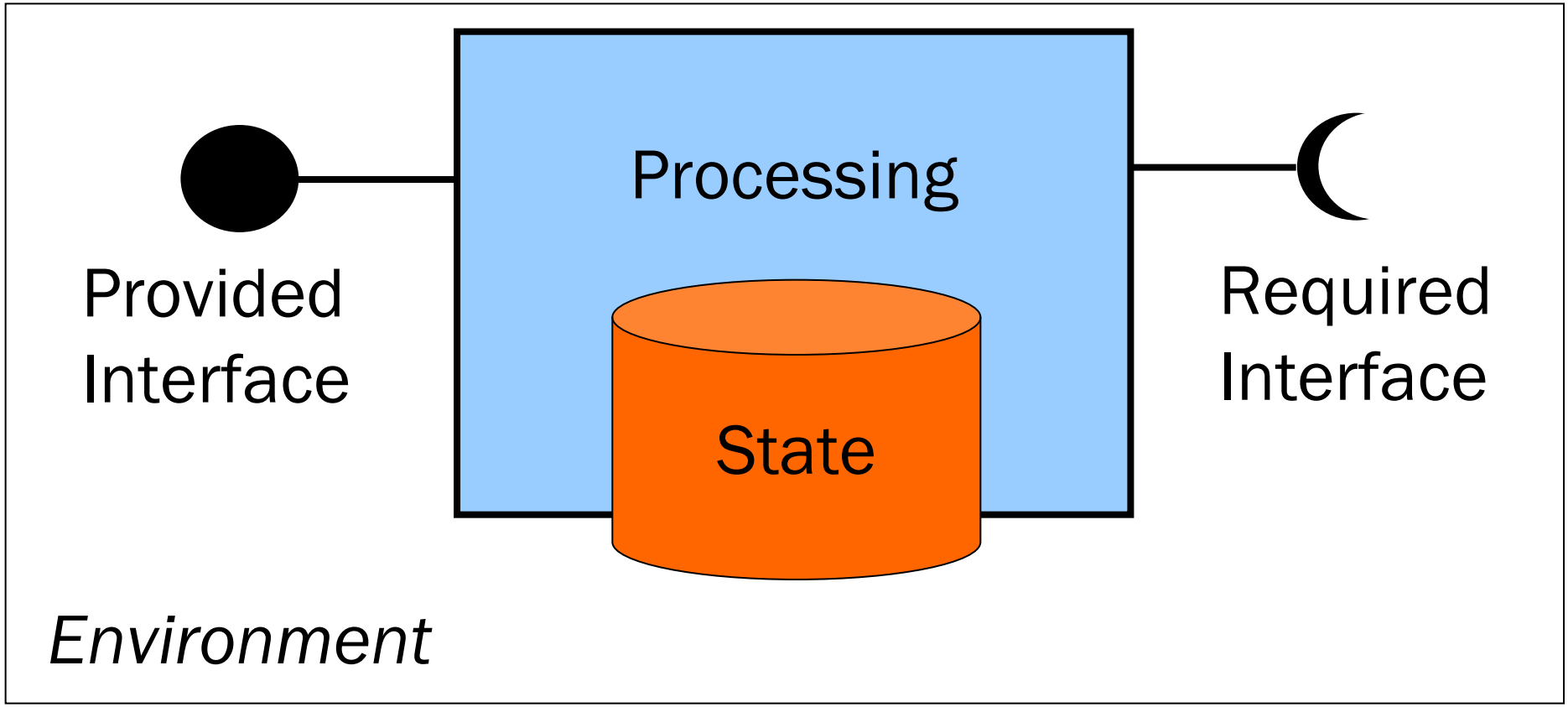


Made out of smaller components

Can be composed into larger systems

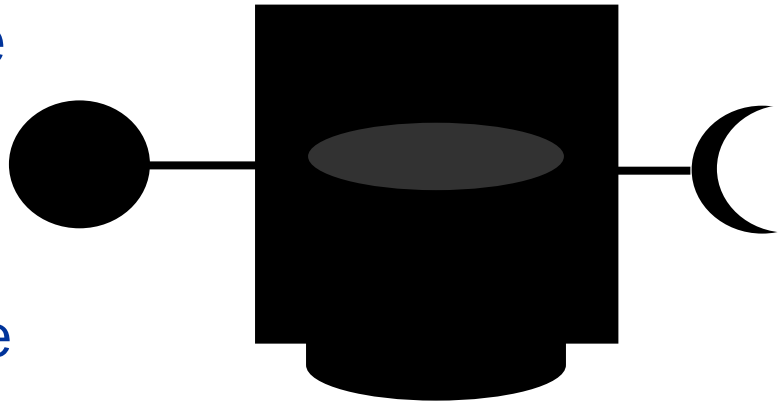
Software Component

- Locus of computation and state in a system



Black Box

- Components only accessible from the “outside”
- Encapsulation
  - Separate the content from the rest of the system
- Abstraction
  - Hide implementation details behind the **interface**
- Modularity
  - Reusable unit of assembly

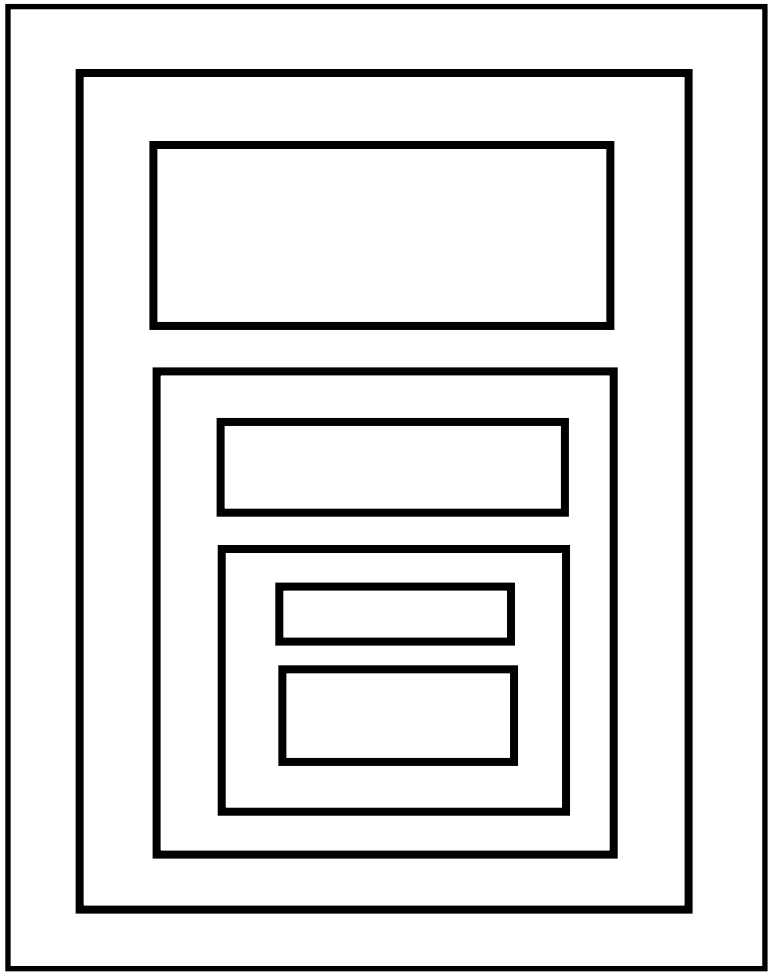


Transparent Boxes

- Black Box - Components only accessible through its interface. Implementation is kept hidden.
- Glass box – Implementation can be inspected, but not modified.
- Grey box – Implementation can be inspected, but only limited modification is allowed
- White box – Implementation can be inspected and modified completely

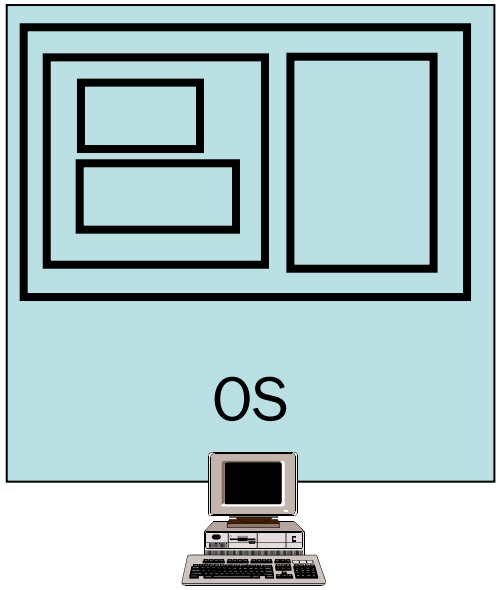
Recursive Components

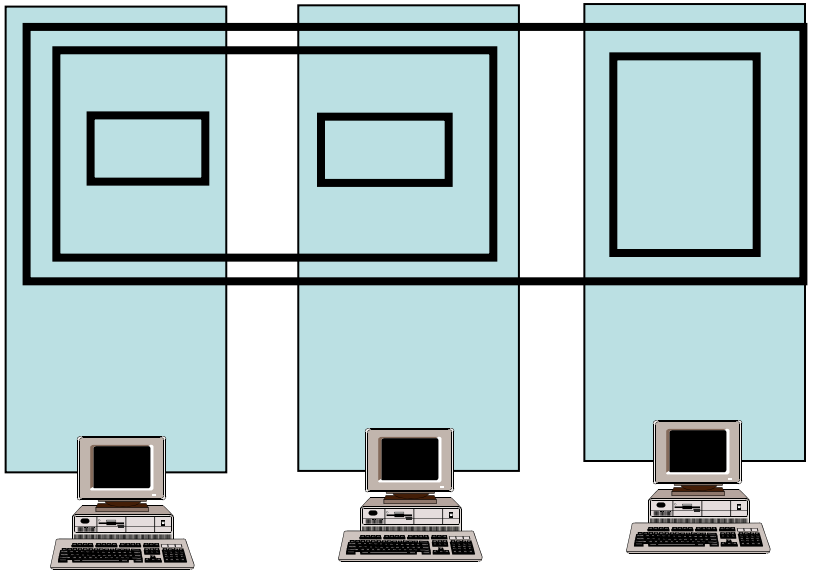
- Components are meant to be composed into larger components
- Components (inside the black box) are made of components
- Software Architecture is Fractal (up to a certain point)



Distributed Components

- Components can be deployed on the same physical host
- Components can be distributed over multiple physical hosts





Kinds of Components

- User-Interface

- Active (Passive)

- Realtime

- Stateful (Stateless)

From John Reekie, Rohan McAdam

Invented by

Doug Mc Ilroy, **MASS PRODUCED SOFTWARE COMPONENTS**, NATO Conference on Software Engineering, Garmisch, Germany, 7-11. October 1968

Software components (routines), to be widely applicable to different machines and users, should be available in families arranged according to precision, robustness, generality and time-space performance. Existing **sources of components** - manufacturers, software houses, users' groups and algorithm collections - lack the breadth of interest or coherence of purpose to assemble more than one or two members of such families, yet software production in the large would be enormously helped by the availability of spectra of **high quality** routines, quite as mechanical design is abetted by the existence of families of structural shapes, screws or resistors. The talk will examine the kinds of variability necessary in software components, ways of producing useful inventories, types of components that are ripe for such standardization, and methods of instituting pilot production.

Component Quality

- In 1996, the maiden flight of the Ariane 5 rocket ended in disaster when the launcher went out of control 40s after take off.
- The problem was due to a reused component from a previous version of the launcher (the Inertial Reference System - SRI) that failed because assumptions made when that component was developed did not hold for Ariane 5.
- The functionality that failed in this component was not required in Ariane 5.

- "data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an exception being thrown… "
- This "lack of protection of this conversion which caused the SRI computer to stop"

Always validate your "trusted components"

http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html

Università della Svizzera italiana

Facoltà di scienze informatiche

Marketplaces

ComponentSource
The Definitive Source of Software Components

Buy Online or Call : 0800 83 5305 Live Help: Click to Chat
Resize to 800x600

日本語サイトへ | Help | Logon | Register | Sign In

Products | Cart | Quotes | Orders | Services | My Account

View in English

Welcome to ComponentSource - The Definitive Source of Software Components

Product Search

Enter search words:

Search

Buy in Swiss Francs (CHF)

We accept credit cards, checks, bank transfers or purchase orders.

Popular Catalogs

Browse over 90 catalogs

Products By Type

- **Components** (1251)
  - .NET (692)
  - ActiveX / COM (424)
  - Java (116)
  - JavaScript / AJAX (55)
  - Flash (9)
  - C++ / MFC (59)
  - DLL (234)
  - VCL (89)
- **Tools** (522)
  - Windows (496)
  - Linux (53)
  - Unix (36)

NetAdvantage for .NET 2008 Volume 1

A set of essential UI controls for Windows Forms and ASP.NET.

- Now fully compatible with Visual Studio 2008 & 2005
- Includes faster, lighter Aikido Controls: New Web UI Framework
- Add a Vista-style user experience to your WinForms application

More info... From $920.38

Best Seller | Top Download

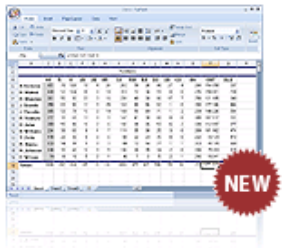
See similar products

Average Review:

New Releases: **Infragistics NetAdvantage for .NET + WPF 2008 Volume 1** - Add WPF (Windows Presentation Foundation) and .NET user interface functionality to your applications.

View All New Releases

Spread for Windows Forms V4.0



Allow developers to easily present, edit and update their tabular data.

ComponentOne Studio Enterprise 2008 v1

Over 180 components for .NET, ASP.NET, WPF, ActiveX + Mobile Devices.

TX Text Control .NET V14.0

Read, edit, and create documents in HTML and Microsoft Word formats.

Categories | View Publishers

3D Modeling (5) Drawing (12) Product Suites (92)

Join Free Today!

- 918,445 members and growing
- Access to Free Products
- Evaluate, Buy & Download 24/7
- Weekly Email Newsletter
- Great Prices & Special Offers

Your email address...

Join Now

Best Sellers

1. NetAdvantage for .NET
2. ActiveReports for .NET
3. DXperience
4. Janus WinForms Controls Suite
5. ComponentOne Studio Enterprise
6. RadControls for ASP.NET
7. NetAdvantage for Windows Forms
8. NetAdvantage for ASP.NET
9. Installshield 2008 Premier
10. GTP.NET

View All Best Sellers

Market Leaders

This week featuring...

janus

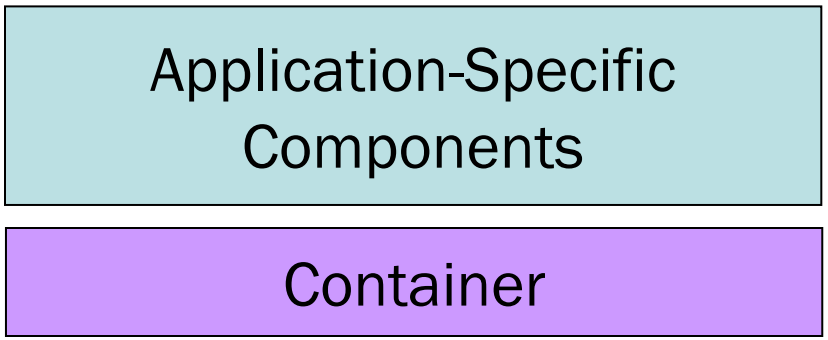
Example Components

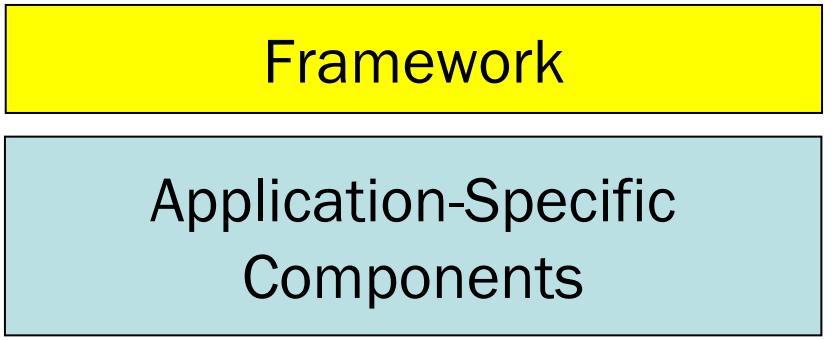
- Application-specific
  - Directly involved in implementing the functionality of the system
  - May be difficult to reuse for other application domains
- Infrastructure
  - Independent of a specific application domain, address the needs of multiple classes of applications
  - Reusable, but may provide much more than what is necessary in one specific application

Component Frameworks

Provide common run-time infrastructure (persistence, distribution, user interface, concurrency/transactions) to support application-specific components

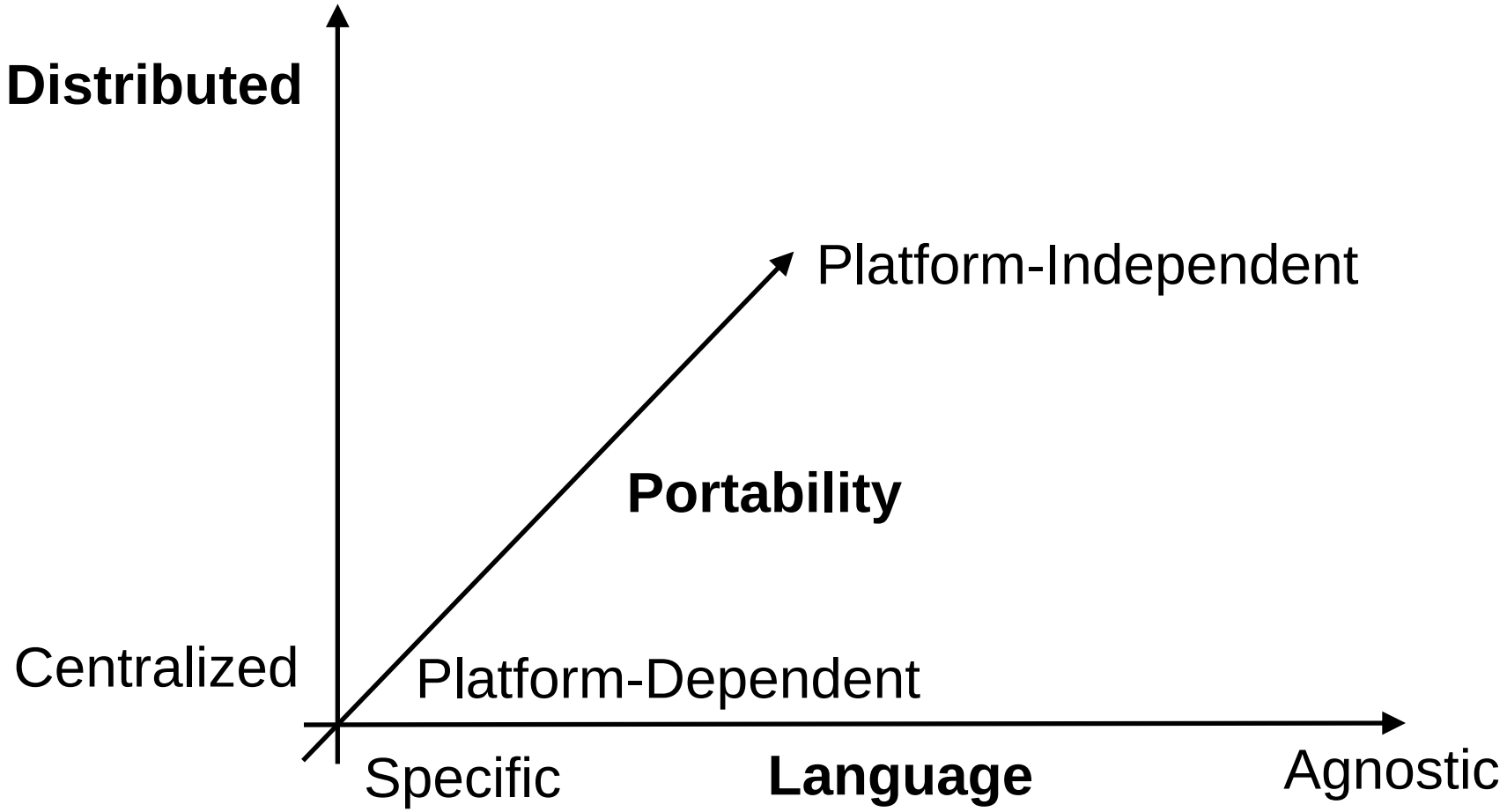
- Component deployed or plugged into the container





- Components "hanging" from the framework (e.g., by inheritance from abstract classes)

Classifying Frameworks



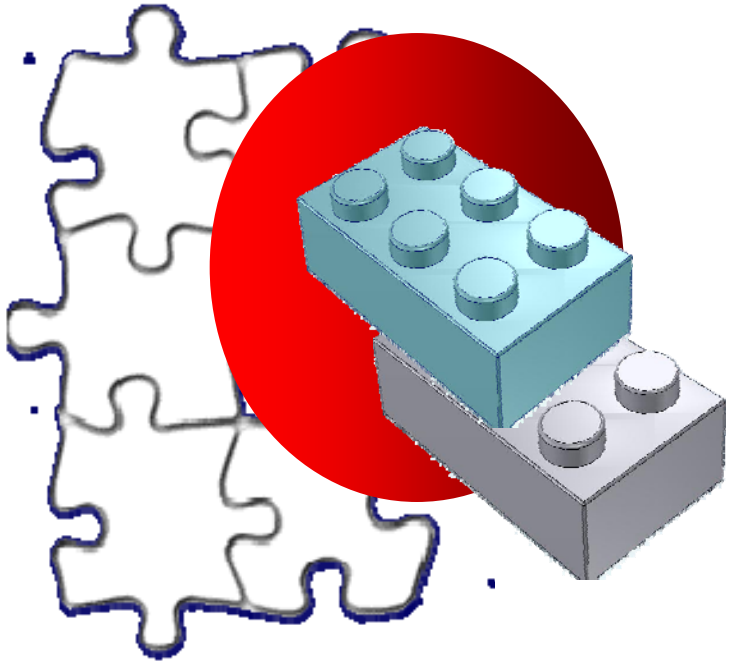
What Component Frameworks do you know?

Ruby on Rails, Eclipse (SWT, JFace, GEF)
AWT/Swing, OMG CORBA, Apache Struts

Microsoft MFC, COM, DCOM, ActiveX, .NET, WCF/WPF
Sun J2EE, Mozilla XPCOM,
GNOME Bonobo, KDE KParts, QT, OpenOffice UNO

Heterogeneity

Due to the lack of interoperability between existing component frameworks, it is not always possible to build a system using heterogeneous components of different frameworks



Tomorrow LAB

Finish Architectural Style Reports

Come to the LAB to get feedback/corrections

Thursday NO Exercise

Università della Svizzera italiana

Facoltà di scienze informatiche

Components, Services, Interfaces and Connectors

Prof. Cesare Pautasso

http://www.pautasso.info

Contents

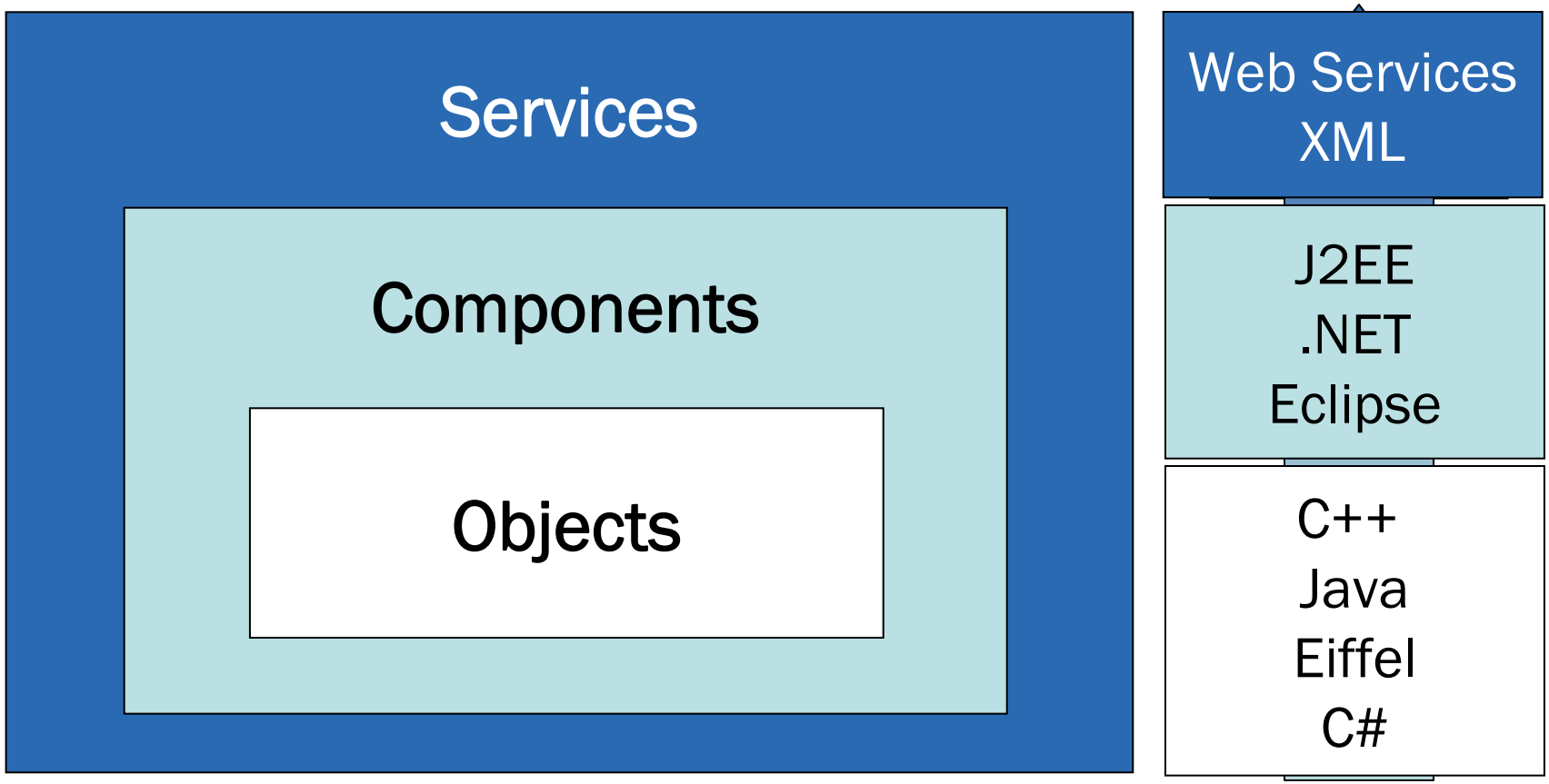
- Components
  - Abstraction Levels: Objects, Components, Services
  - Interoperability
  - Interfaces
- Connectors
  - Coupling Dimensions

CBSE Principles

Component Based Software Engineering

- Components are independent unit of deployment and composition that should not interfere with each other (*but can explicitly require other components)*
- Component as black boxes:
  - Components communicate through well-defined (and documented/standardized) public interfaces
  - Implementation details are hidden behind the interface

Abstractions Levels



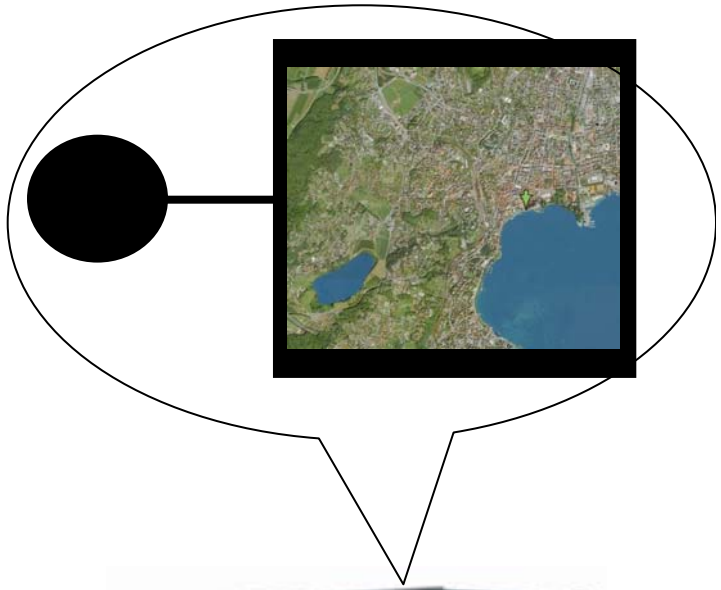
Component vs. Objects

- Component
  - Encapsulate state and functionality
  - Coarse-grained
  - **Reusable unit of composition**
  - Well-defined (documented, standardized) interface contract with explicit dependencies
  - Architecture Element: Implementation not important (Black Box)
  - **High Quality**
- Object
  - Encapsulate state and functionality
  - Fine-grained
  - **Identifiable unit of instantiation**
  - May not always provide a public interface
  - Hard to reuse by itself (lots of dependencies to other objects)
  - At run-time can “move” between components
  - Object-Oriented Programming Language Typing Construct

Components vs. Services Example

Map Drawing Component

```
Map getMap(lat, long, zoom)
```



- (`lat, long`) centers the map on any location on the planet Earth and `zoom` can show details up to 1m precision

- What is the "size" of this component?
- How to deliver the component to customers so that it can be included in their own applications?

Components vs. Services Business Model

- How to sell a component?
  - Component developers charge on a **per-deployment** basis: whenever a new client downloads the component.
  - Component **upgrades** may be sold separately to generate a revenue stream
  - Components can be licensed to be redistributed within larger systems and developers can demand **royalties** from the revenue of the final product
- How to sell a service?
  - Service providers can charge on a **per-call** basis: each time an existing client interacts with a service by exchanging a new message.
  - Service providers can charge a monthly/yearly flat **access fee**
  - Services can be made available for free and providers can support them with **advertising** revenue

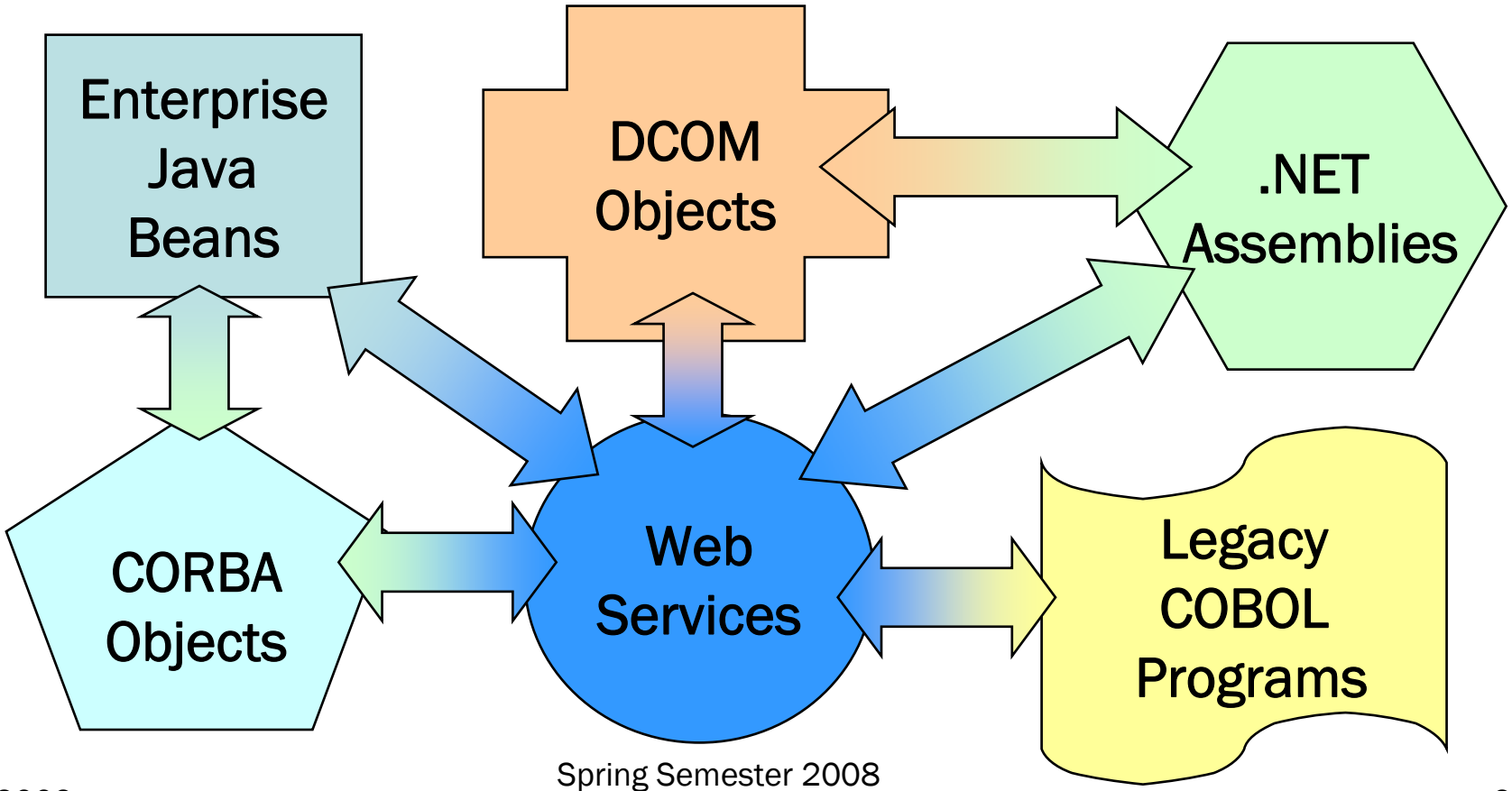
Components vs. Services Technology

- To be used a component must:
  - be packaged to be deployed as part of some larger application system
  - fit with the existing framework used to develop the system
- There are many component frameworks available for building distributed systems (e.g., J2EE, DCOM, .NET, CORBA).
- The problem is: they are not compatible (as an exercise try to use a .NET assembly to make an Eclipse plug-in and see what happens)

- To be used a service must:
  - be published on the Web (once)
  - advertise its description and location to potential clients across the Web so that they can access it using standard protocols
- Like components, services can be reused, composed into larger systems and (of course) they can be found on the Web.
- Unlike components, services do not have to be downloaded and deployed in order to be used by clients. Instead, a client may discover and access their functionality by using standard protocols (WSDL, SOAP, UDDI) based on XML standards.

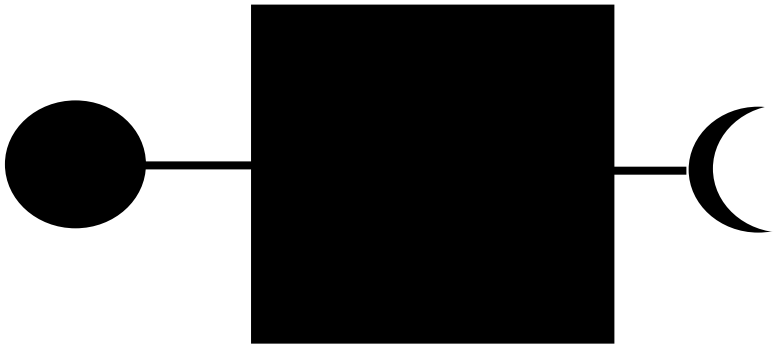
Component Interoperability

- Due to lack of interoperability, it is not always possible to build a system using heterogeneous components



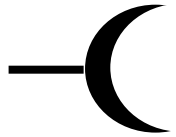
Interfaces

- Specify and document the externally visible features (or the public API) of the component
- **Operations**
  - Call functionality, modify state
- **Properties**
  - Read visible state attributes
- **Events**
  - "Call Backs"
- **Dependencies**
  - What the component requires

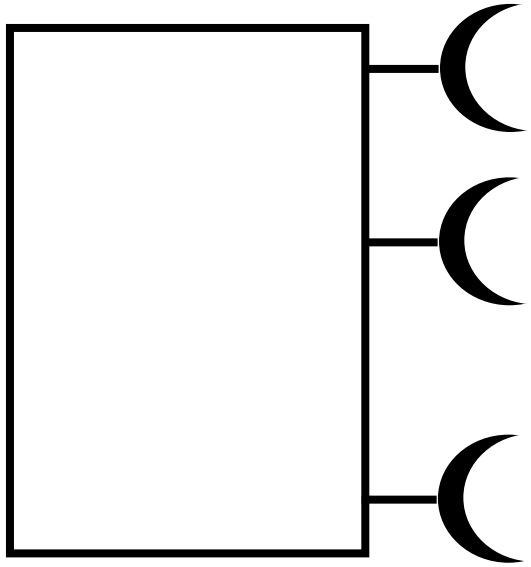


- **Information Hiding Principle:** always keep the implementation details *secret* (invisible from the outside) and only access them through the public interface

Dependencies



- Which assumptions makes a component?
- A component can be reused if:



The *required interface* is satisfied
(by other components that provide it)

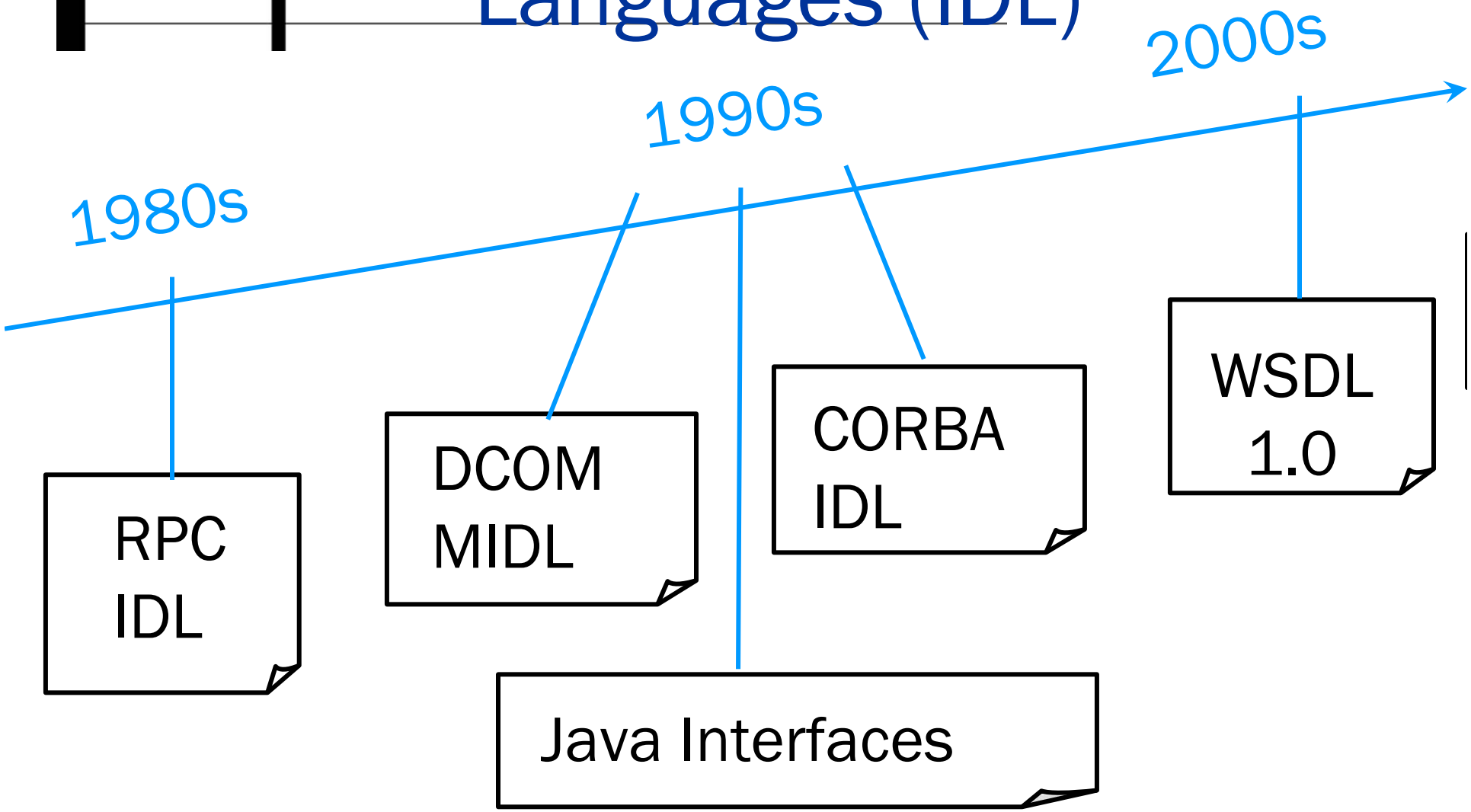
The *platform* is compatible

- Runtime Libraries/Framework
- Operating System/Device Drivers
- Hardware

The *environment* is setup correctly:

- Databases
- Configuration Files
- File System Directories

Interface Description Languages (IDL)



Interface Design Principles

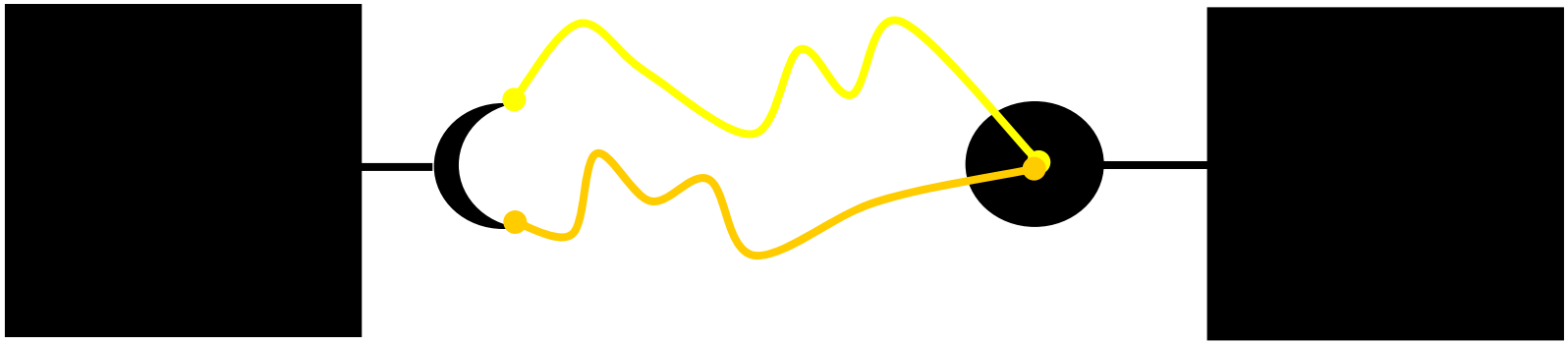
- **Explicit Interfaces**
  - Components always communicate through interfaces
- **Uniform Access**
  - Facilities managed by a component are accessible to its clients in the same way whether implemented by computation or by storage.
- **Few Interfaces**
  - Every component communicates with as few others as possible
- **Clear Interfaces**
  - Do not publish useless functionality (unless it is used by other components)
- **Small Interface**
  - If two components communicate, they exchange as little information as possible

From Bertrand Meyer

Reusable Interfaces

- Usability vs. Generality
  - A general interface helps to reuse a component in many architectures
  - A general interface is more complex and difficult to use in one architecture
- Performance
  - A general reusable component may be less optimized and provide worse performance than a specific, non-reusable one
  - For the same interface, there can be multiple implementation components optimized for different hardware platforms
- Legacy "Wrapping"
  - Existing system can be reused as components by adding a standard interface that does not require to modify them (which can be very expensive or impossible to do safely)
- Minimize Required Interface
  - Self-contained (zero dependencies) components are most reusable

From Ian Sommerville

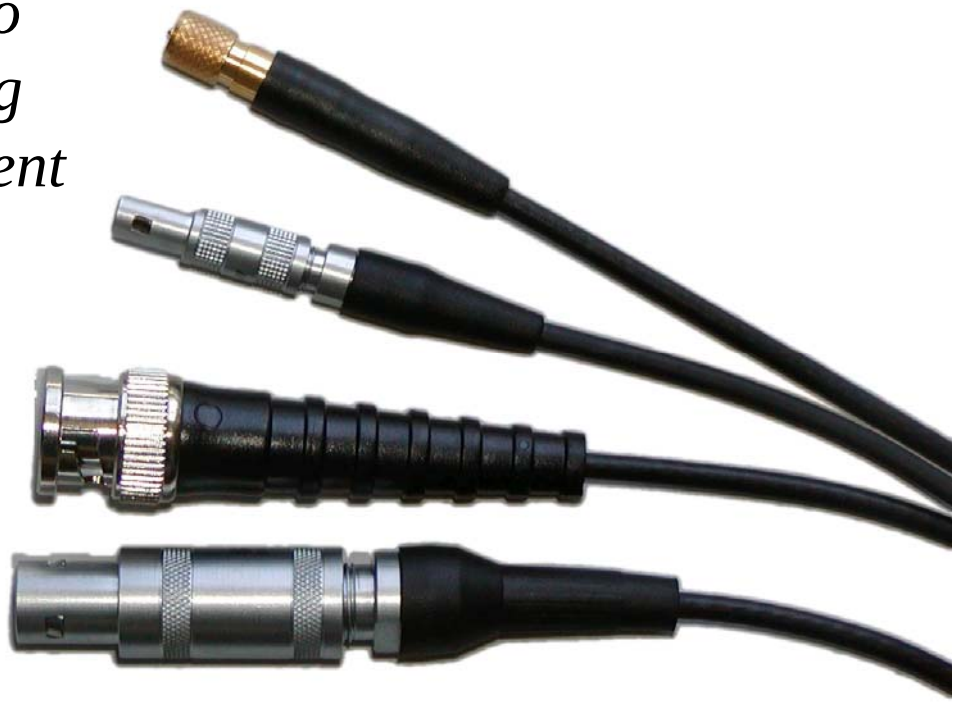


Connectors

Connector

- Generic enabler of composition

Plug into matching component ports

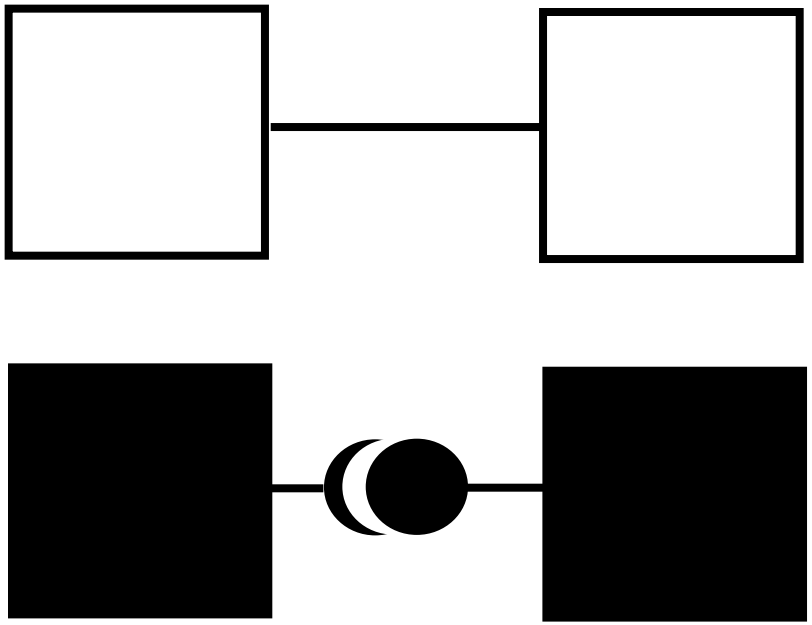


Transfer signals (data, control) between ports

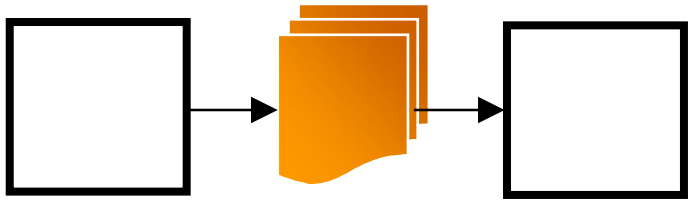
Connectors enable architects and engineers to assemble heterogeneous functionality, developed at different times, in different locations, by different organizations.

Software Connector

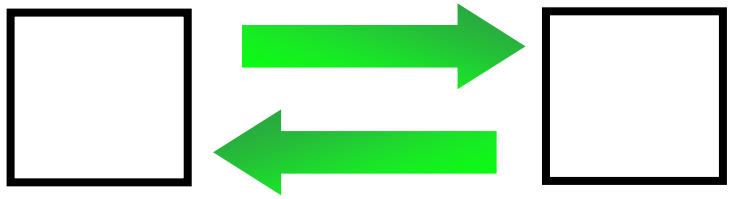
- A software connector is the architectural element tasked with effecting and regulating **interactions** among components
- A connector couples two or more components to perform transfer of data and control
- Component interfaces need to “match” to be connected



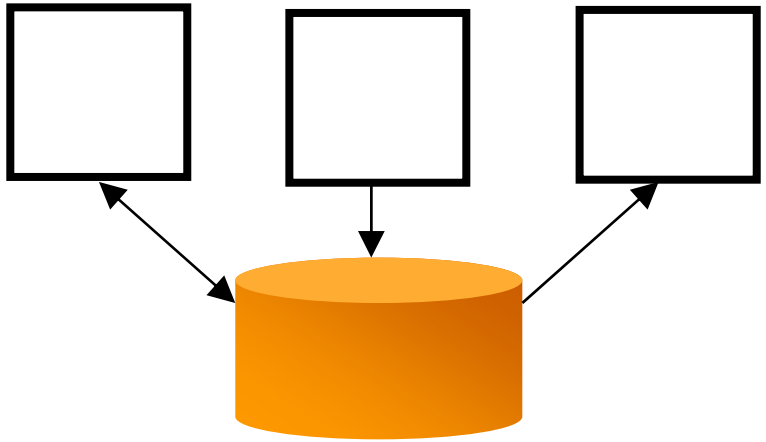
Connector Examples



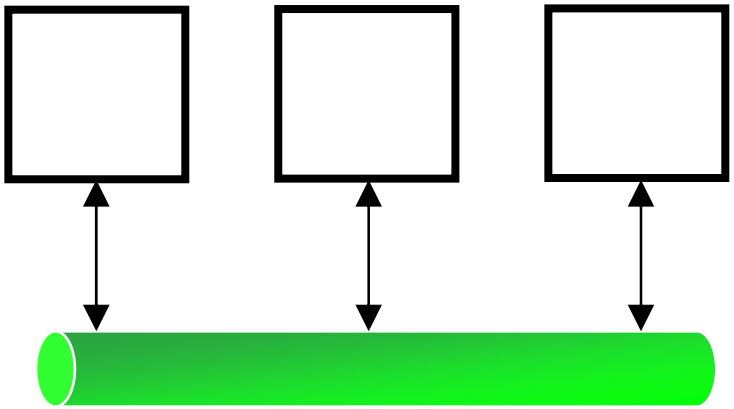
File Transfer



Procedure Call
Remote Procedure Call



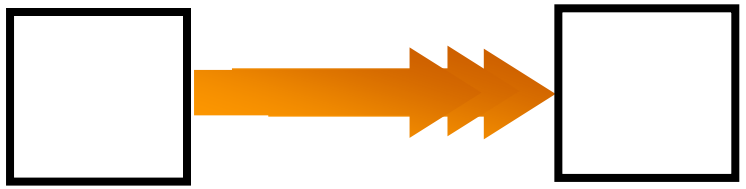
Shared Data



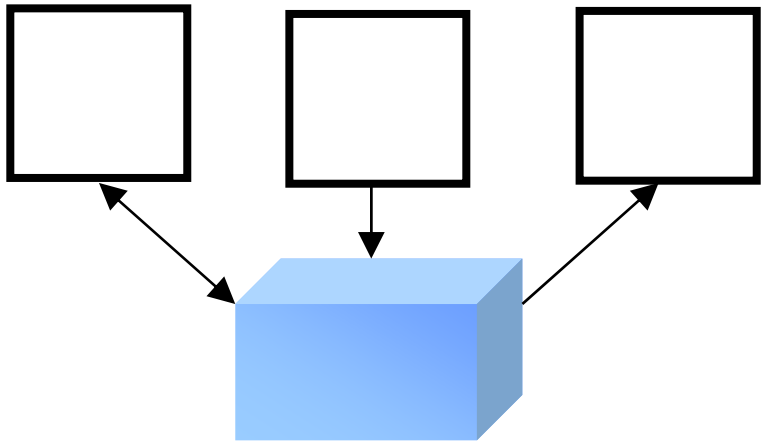
Message Bus
Events

Connector Examples

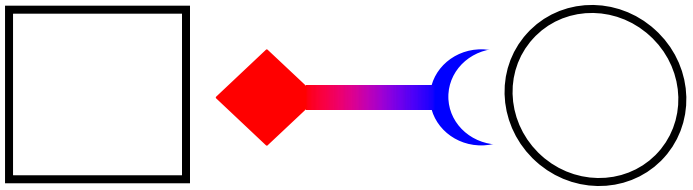
Linkage
(Static/Dynamic)



Stream

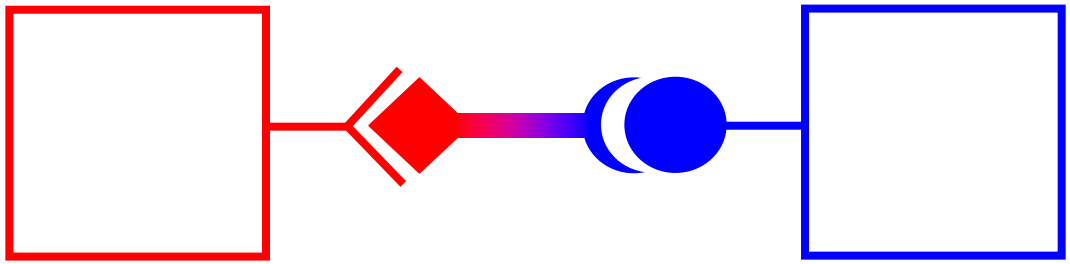


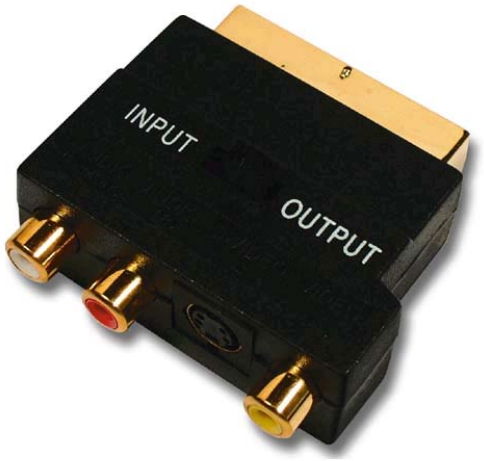
Arbitrator



Adapter

Adapter



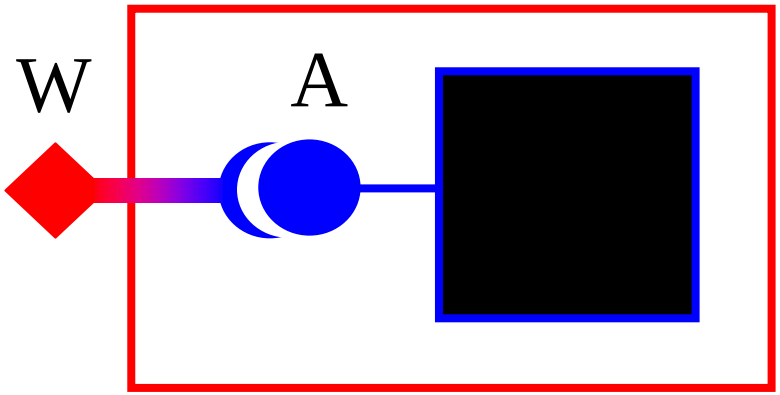


- Architectures may choose to reuse pre-existing components that do not always "fit" perfectly
- Adapters help to connect mismatching interfaces by doing the necessary conversion

- Mismatches:
  - Naming
  - Typing
  - Syntax/Structure
  - Granularity
  - Interaction style
  - Semantics
  - Missing data/functionality

Wrapper

- Wrapping is a pattern to reuse components with incompatible interface
- The interface A is hidden “inside” a component with interface W implemented with the Adapter connector



Tomorrow Lab
Thursday Exercise (SI 013)

Università della Svizzera italiana

Facoltà di scienze informatiche

Modeling Architectures

Prof. Cesare Pautasso

http://www.pautasso.info

Università della Svizzera italiana

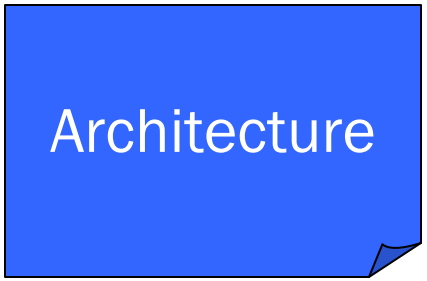
Facoltà di scienze informatiche

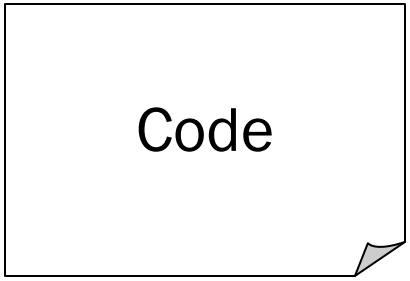
Contents

- Models of Architecture
- Modeling Process
- Multiple Views
  - 4+1
- Quality of Models
- Modeling Techniques Overview

Capturing the Architecture

- **Every system has an architecture**
  some architectures are made manifest and visible, many others are not
- A system's (descriptive) architecture ultimately resides in its executable code
- Before a system is built, its (prescriptive) architecture should be made explicit
- A system's architecture may be **visualized and represented using models** that are somehow related to the code (existing or to be written)





From Grady Booch

Why Modeling?

- **Record** decisions
  - Which decisions? What level of detail?
  - Document the architecture
- **Communicate** decisions
  - Notations (Visualization)
  - Different roles involved in the project
- **Evaluate** decisions
  - What is a good architectural model?
  - Help to detect problems early
- **Evolve** decisions
  - Give constraints and a clear path to change the system

- **Generate artifacts**
  - With detailed enough models, it is possible to drive the development of the code from the model itself

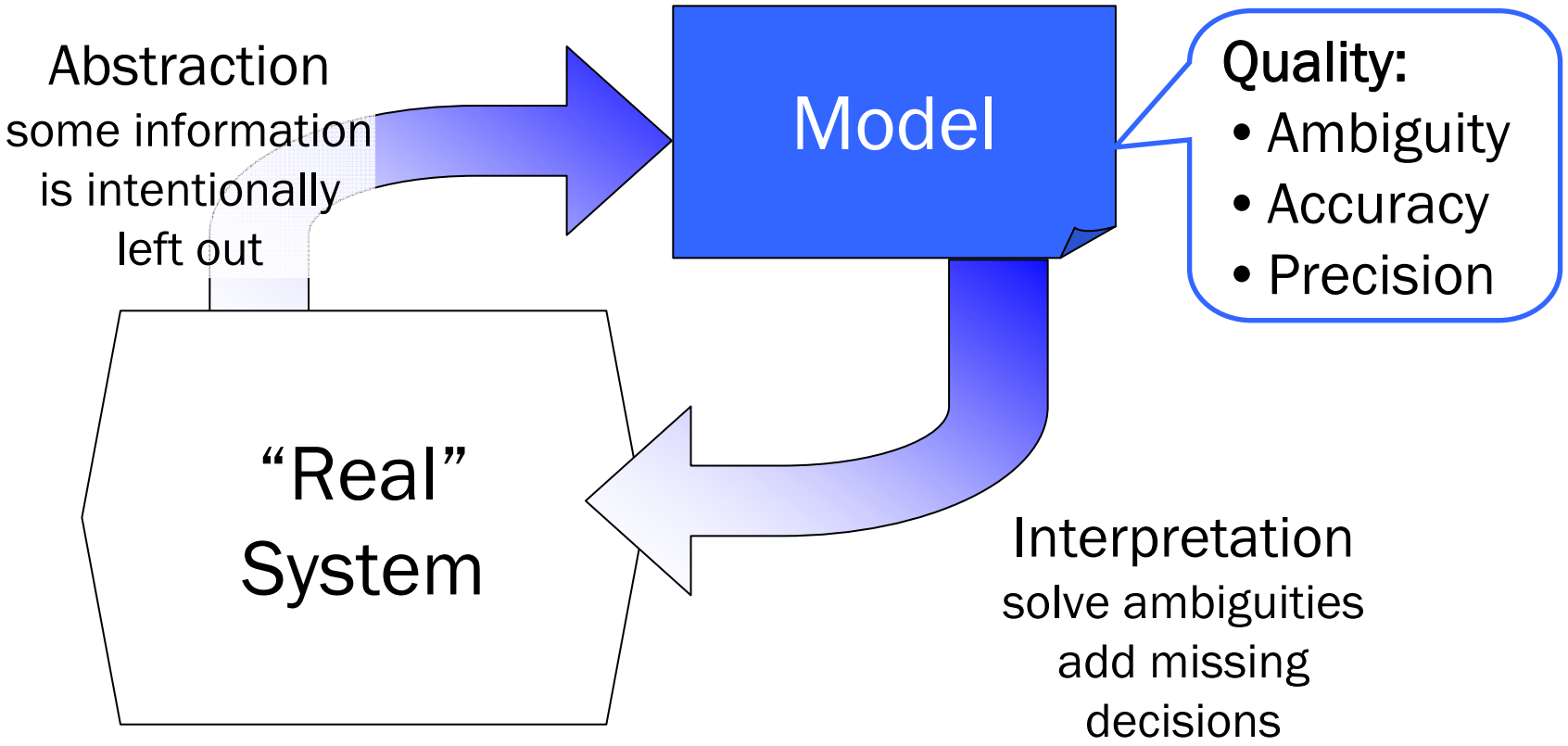
Definitions

- An architectural *model* is an artifact that captures some or all of the design decisions that comprise a system's architecture.
- Architectural *modeling* is the reification and documentation of those design decisions.

From Richard Taylor et al.

Abstraction and Interpretation

- The architecture models only some interesting aspects of a system



Modeling Process

1. What aspects of the architecture to model?
2. What is their importance for the project?
3. What is the goal of the model?
4. Select suitable notation and level of detail
5. Design good models
6. Use the models consistent with their goal

From Richard Taylor et al.

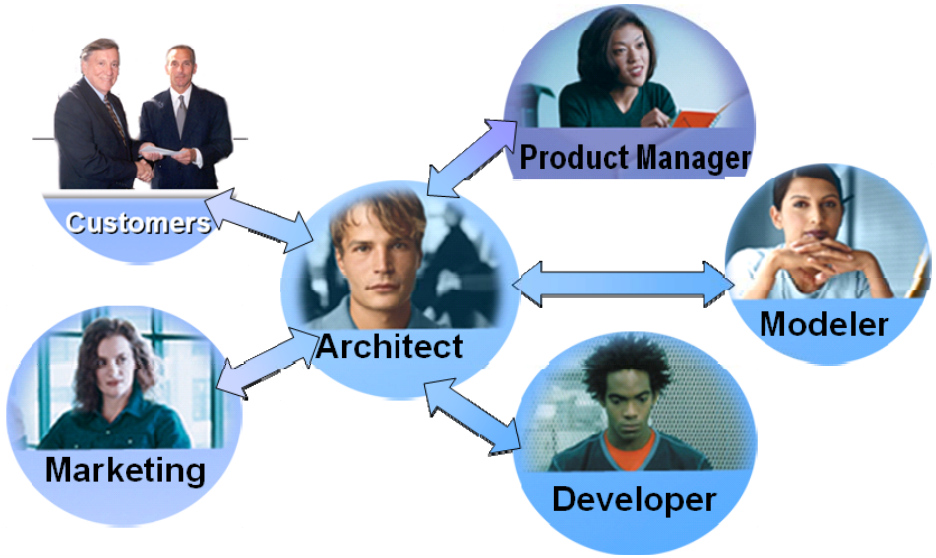
1. What to model?

- Static Architecture:
  - Structural Decomposition
  - Interfaces
  - Components
  - Connectors
  - Mapping to Code Artifacts
- Dynamic Architecture:
  - Behavior
  - Deployment
  - Mapping to Hardware
- Design Process:
  - Rationale of decisions
  - Stylistic Constraints
  - Dependencies on other projects
  - Team Organization
  - Legal Constraints
- Quality:
  - Non Functional Properties
  - Testing

2. How much modeling?

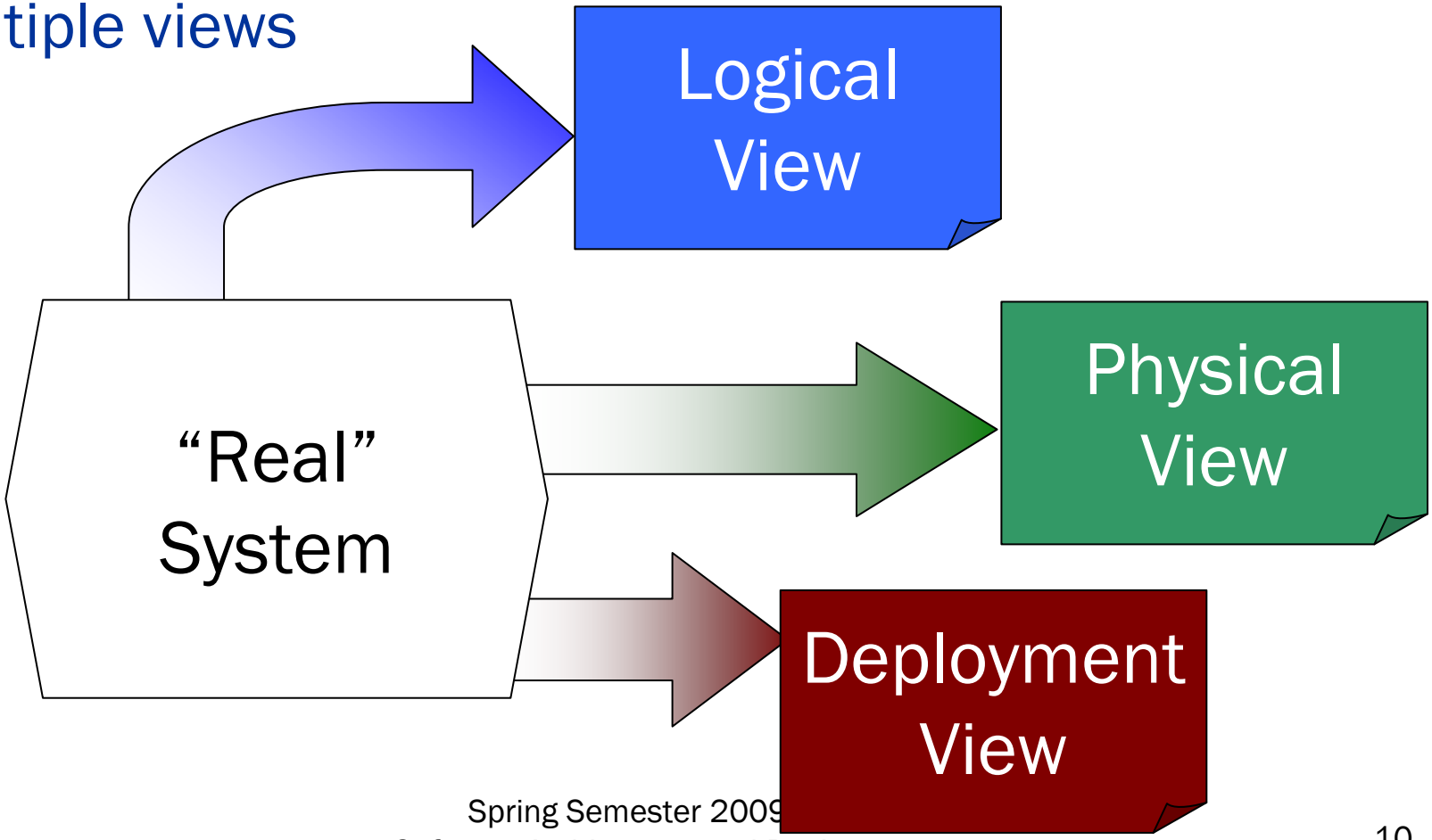
- Scope of the model:
  - The architecture of the entire system
  - Parts of the system for different use cases
  - The style of the system (constrain the actual architecture without describing it)
- Add details to the model only where they are needed

- Target Audience:
  - Technical Developers
  - Marketing/Customers
  - Management



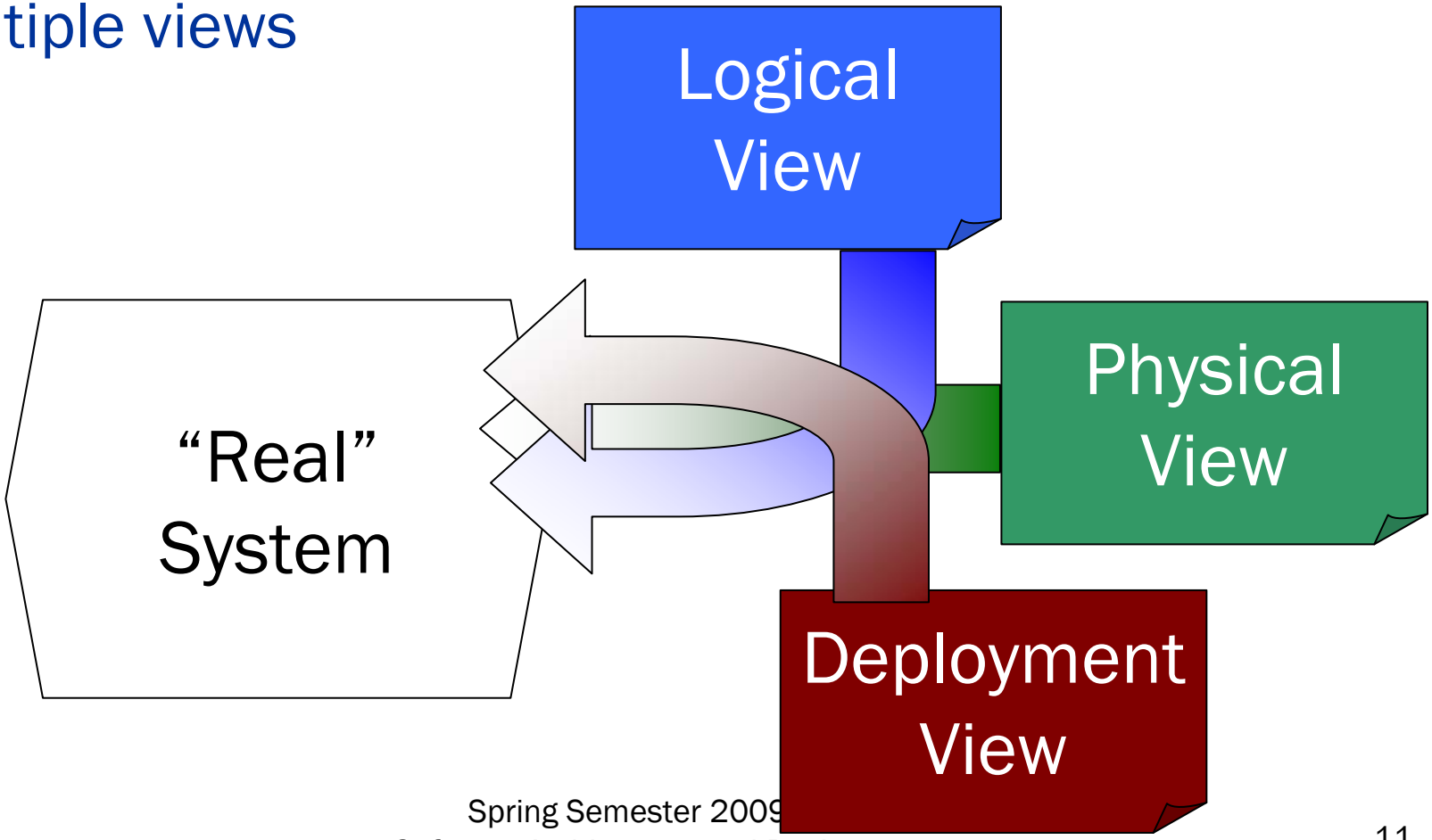
Multiple Views

- There is too much information to model, we need multiple views



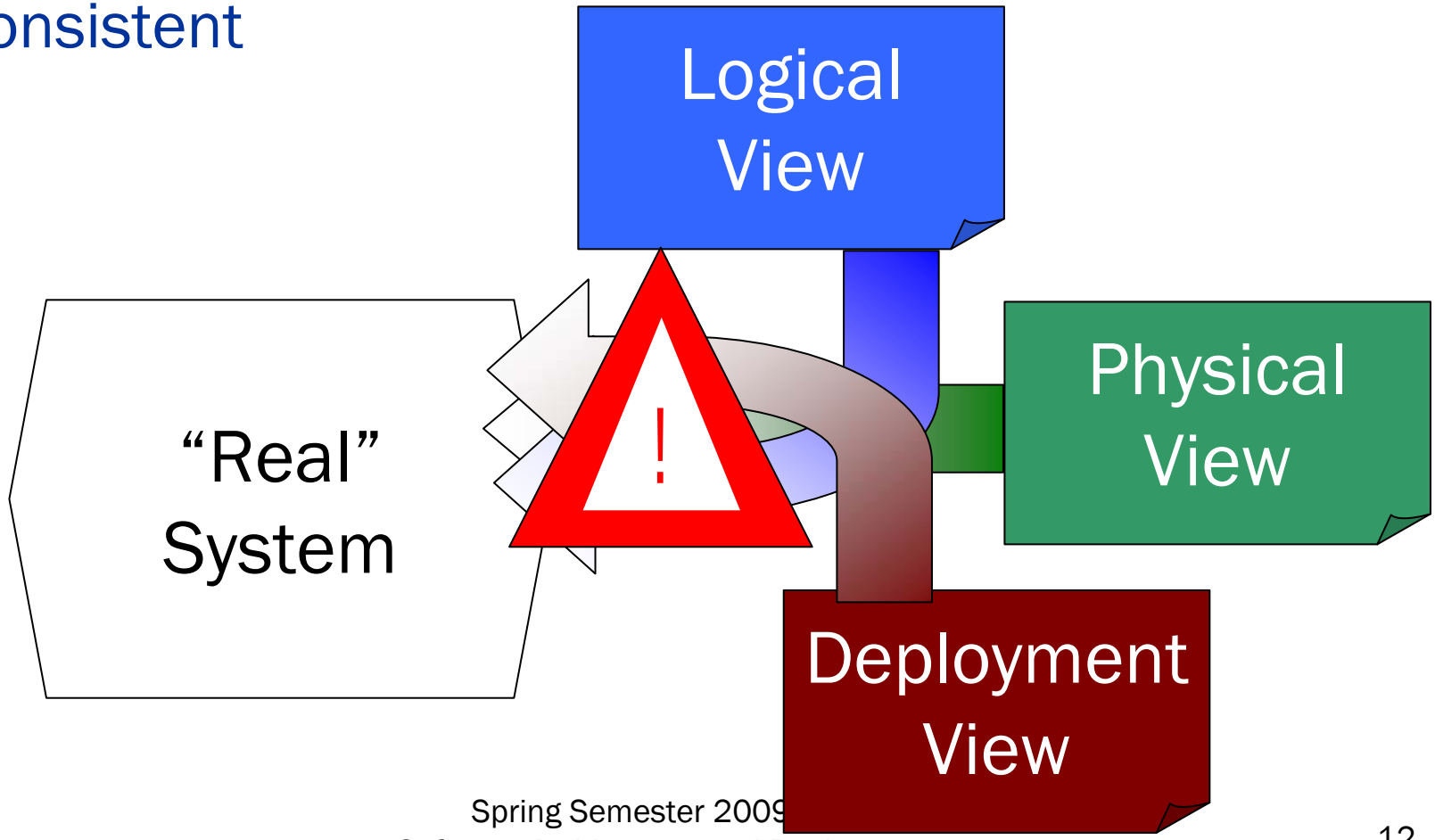
Multiple Views

- There is too much information to model, we need multiple views



Consistency Problem

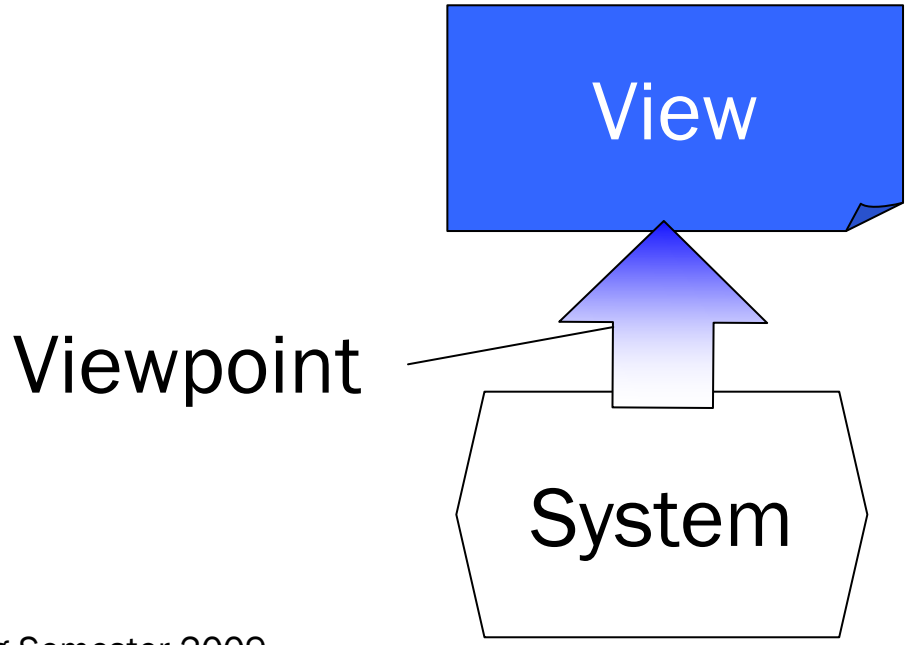
- Views are not orthogonal and should not become inconsistent



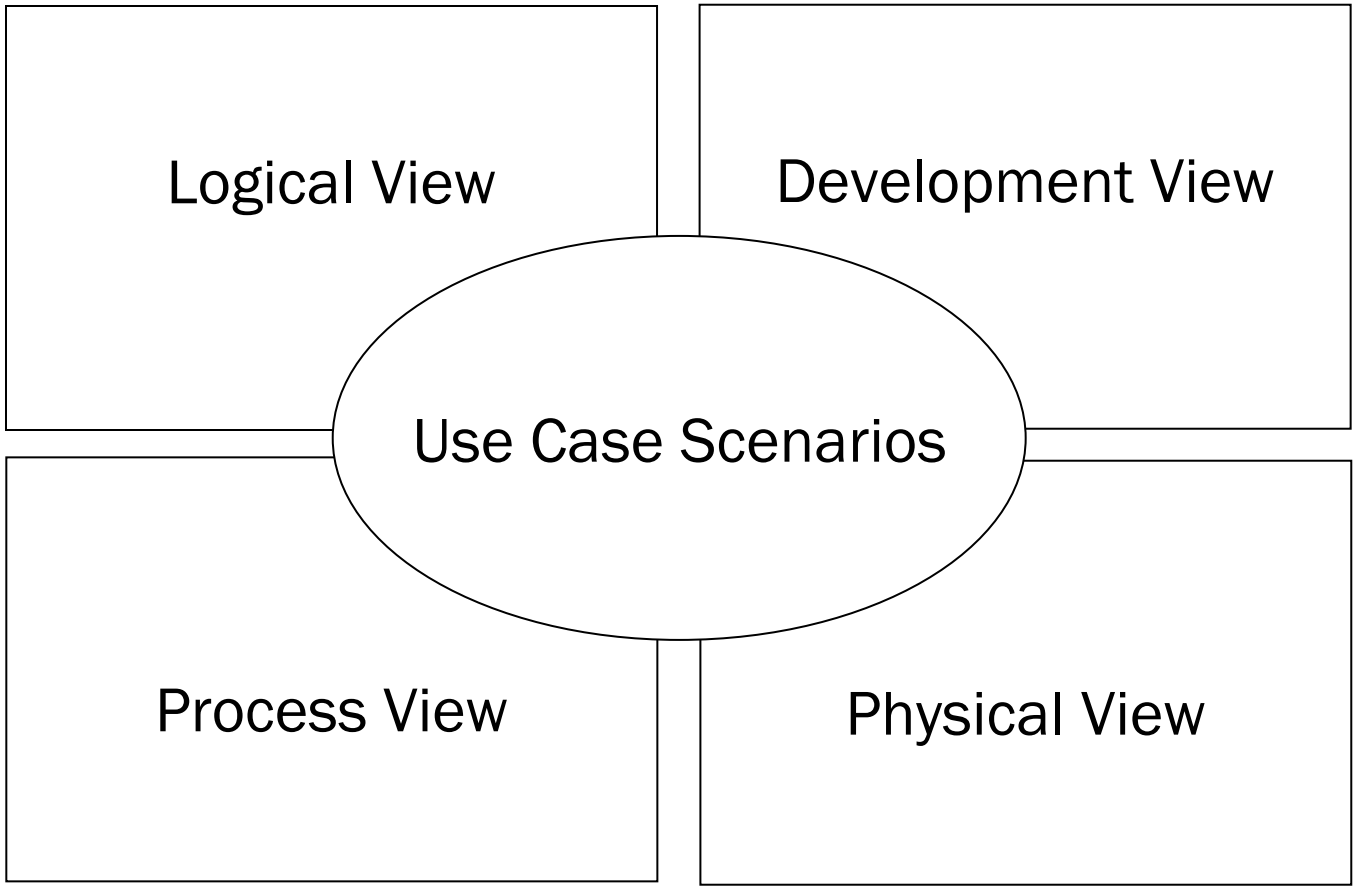
What is a View?

- No single modeling approach can capture the entire complexity of a software architecture
- Various parts of the architecture (or views) may have to be modeled with a different:
  - Notation
  - Level of detail
  - Target Audience

- A *view* is a set of design decisions related by common concerns (the *viewpoint*)

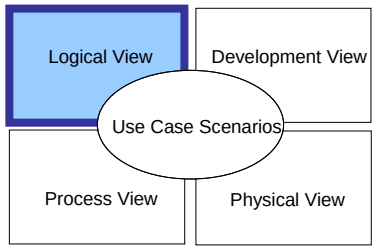


4+1 View Model



From Philippe Kruchten

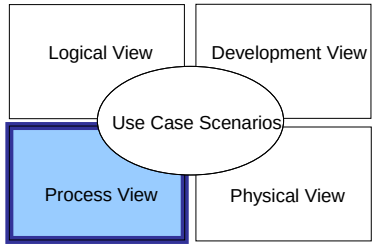
Logical View



- Decompose the system structure into software components and connectors
- Map functionality/requirements/use cases onto the components
- Concern: Functionality
- Target Audience: Developers and Users

From Philippe Kruchten

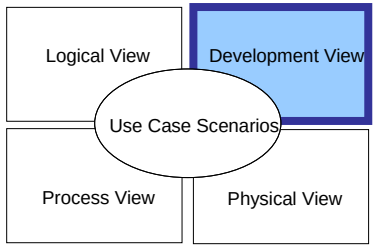
Process View



- Model the dynamic aspects of the architecture:
  - Which are the active components?
  - Are there concurrent threads of control?
  - Are there multiple distributed processes in the system?
  - What is the behavior of (parts of) the system?
- Describe how processes/threads communicate (e.g., RPC, Messaging connectors)
- Concern: Functionality, Performance
- Target Audience: Developers

From Philippe Kruchten

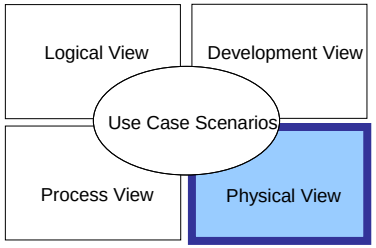
Development View



- Static organization of the software code artifacts (packages, modules, binaries…)
- A mapping between the logical view and the code is also required
- Concern: Reuse, Portability, Build
- Target Audience: Developers

From Philippe Kruchten

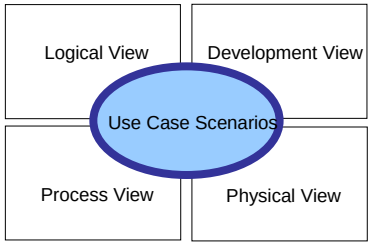
Physical View



- Define the hardware environment (nodes, hosts, networks, storage, etc.) where the software will be deployed
- Different hardware configurations may be used for providing different qualities
- A Mapping between logical and physical entities is also necessary (sometimes found in a separate Deployment View)
- Concern: Performance, Scalability, Availability, Reliability
- Target Audience: Operations

From Philippe Kruchten

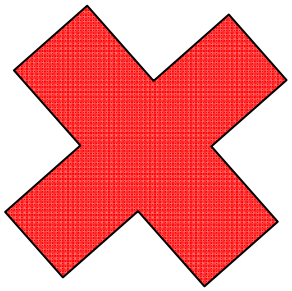
Scenarios



- The modeling elements of the 4 views are linked by scenarios that show how they work together to fulfill the use cases of the system
- The actual model of the architecture can be broken down in scenarios, that are illustrated using the notations of the 4 views
- Scenarios help to ensure that the architectural model is complete with respect to requirements
- Scenarios can be prioritized to help driving the development of the system

From Philippe Kruchten

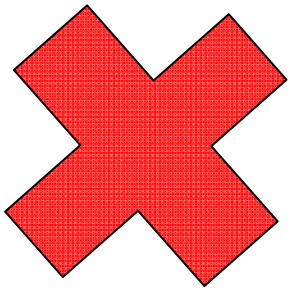
What is a good model?



We designed it with UML

therefore

it is a good architecture



- Improving the quality of a model is done by making **good** design decisions and documenting those decisions with adequate precision using the most suitable notation (not the other way around).

Model Quality

- Ambiguity
  - A model is *ambiguous* if it leads to more than one interpretation
  - Incomplete models can be ambiguous: different people will fill in the gaps in different ways.
- Accuracy
  - A model is *accurate* if it is **correct**, conforms to fact, or deviates from correctness within acceptable limits.
- Precision
  - A model is *precise* if it is sharply exact or delimited.

- Make sure your architecture is accurate (a wrong or conflicting architectural decision is a recipe for disaster)
- Sometimes you can even make it complete (but it will be more expensive, so only do it for critical aspects of the system)
- Precision helps, but/if you can trust developers to add detail (avoid over-specifying and over-designing the architecture, especially if the architecture is inaccurate, adding details will not fix it)

Accuracy and Precision

Inaccurate
Imprecise

(a)

(b)

Accurate but imprecise

Very precise but inaccurate

(c)

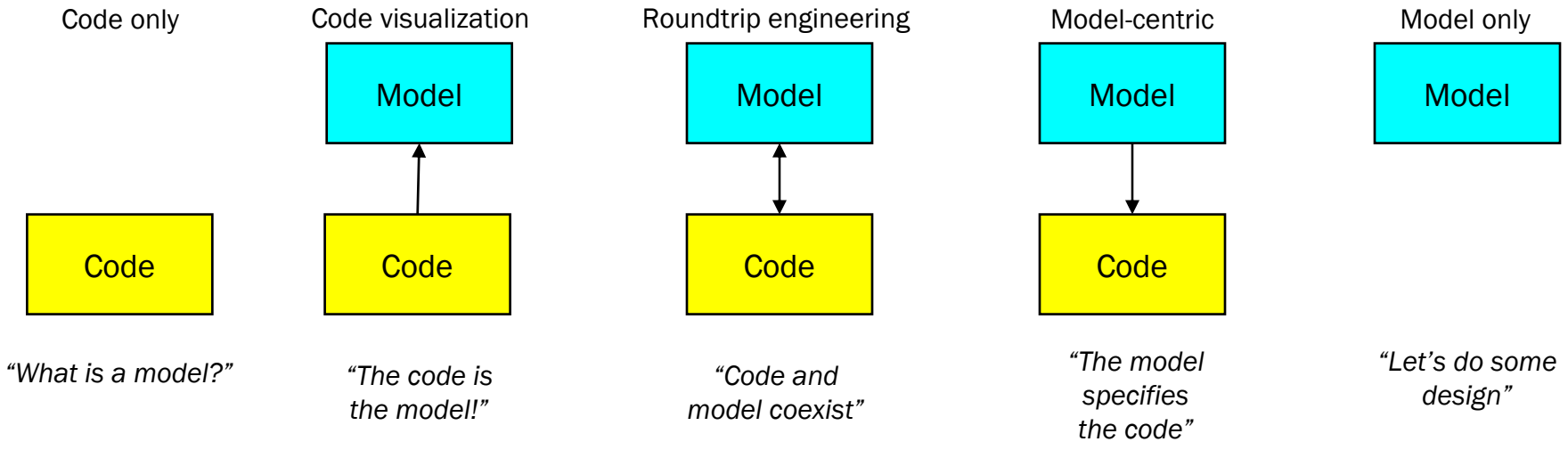
(d)

Accurate and precise

From Richard Taylor et al.

Model-Driven Engineering

- MDE is a style of software development centered on modeling
- Design of systems is organized around a set of models and **model transformations** to move within and between different abstraction layers (e.g., code generation, reverse engineering)



From IBM Rational

Università della Svizzera italiana

Facoltà di scienze informatiche

Unified Modeling Language for Architects

Prof. Cesare Pautasso

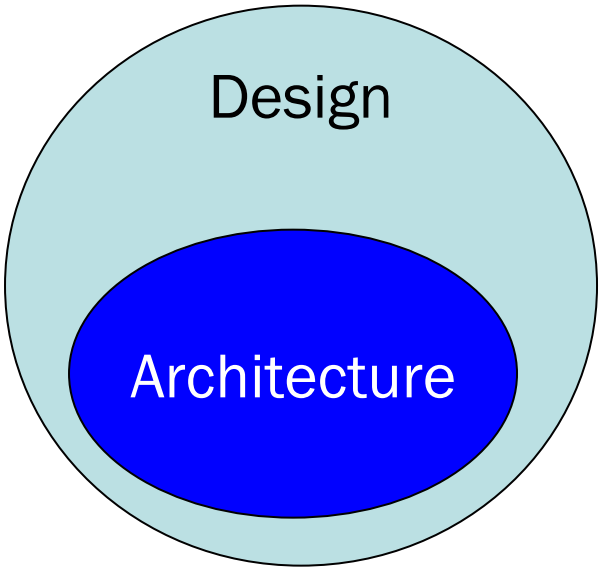
http://www.pautasso.info

Contents

- Modeling Techniques Overview
- Unified Modeling Language
  - Component and Deployment Diagrams
  - Interaction (Activity and Sequence) Diagrams
  - Class Diagrams
  - State Machine Diagrams
  - Use Case Diagrams

How much modeling effort?

- All architecture is design, but not all design is architecture
- Architecture focuses on *significant* design decisions, decisions that are both structurally and behaviorally important as well as those that have a lasting impact on the performance, reliability, cost, and resilience of the system
- Architecture involves the how and the why, not just the what



From Grady Booch

Maturity of Modeling Techniques

- Ideally, stakeholders and architects would simply be able to *select a mature notation that would cover their modeling and analysis needs*
- However, the reality is that there are not enough mature notations currently available to make this possible. Even if one were prepared to use several notations simultaneously (and deal with the costs of managing consistency among them), there will likely still be gaps between what can be modeled and what stakeholders need.
- This means that architects will not only have to invest in modeling, but invest in **developing the technologies they use to model**.

From Richard Taylor et al.

Modeling Techniques

- Natural Language
- PowerPoint
- Unified Modeling Language (UML)
- Architectural Description Languages (ADL)

Evaluation Criteria

Scope and Purpose	What is it intended to model with the technique and what is it *not* intended to model?
Basic Elements	What are the basic modeling elements and concepts?
Style	Can the technique capture stylistic constraints? For what kind of styles?
Static and Dynamic Aspects	Does the technique only support structural modeling or can also model behavior?
Dynamic Modeling	To what extent can the model be changed to reflect changes as a system executes?
Non-Functional Aspects	Does the technique include explicit support for capturing or evaluating the quality of the system
Ambiguity	Are multiple interpretations of the model possible or disallowed?
Accuracy	Can the correctness of the model be determined using the technique?
Precision	Does the technique allow to refine the model to higher detail levels?
Viewpoints	Can a model built using the technique be partitioned into multiple views?
View Consistency	Does the technique help to detect and deal with inconsistencies across multiple views?

Natural Language, PowerPoint

Evaluation

Text (Natural Language)

Scope and Purpose	Describe all aspects of an architecture informally
Basic Elements	Extensive vocabulary
Style	Style can be captured through generalizations
Static and Dynamic Aspects	Both
Dynamic Modeling	Models can be manually rewritten but cannot be tied to the implementation automatically
Non-Functional Aspects	Can be described but not verified automatically
Ambiguity	Most ambiguous (predefined vocabularies and templates can help to reduce ambiguity)
Accuracy	Checked with manual inspection
Precision	Just add more text to add details
Viewpoints	All viewpoints (treated in the same way)
View Consistency	Checked with manual review

PowerPoint (Slideware)

Scope and Purpose	Visualize all aspects of an architecture informally through arbitrary diagrams and text
Basic Elements	Geometric shapes and lines, text, clip art, animations
Style	Not supported
Static and Dynamic Aspects	No semantics
Dynamic Modeling	No support
Non-Functional Aspects	Informal decorations/text on diagrams
Ambiguity	Predefined shapes and templates can help to reduce ambiguity
Accuracy	Checked with manual inspection
Precision	How much information fits on one slide?
Viewpoints	All viewpoints (no direct support)
View Consistency	Checked with manual review

Unified Modeling Language

http://www.uml.org/

Evaluating UML

Scope and Purpose	Originally focused on low-level object-oriented design, in version 2.0 extended to model architectural-level decisions (13 diagram types).
Basic Elements	Classes, Associations, States, Events, Transitions, Activities, Messages, Components, Connectors, Ports, Lifelines, Frames, Actors, Use Cases, Deployment Nodes, Constraints
Style	Captured as OCL Constraints or Profiles
Static and Dynamic Aspects	Static (Class, Package, Object, Component Diagram) Dynamic (State, Activity, Sequence Diagrams)
Dynamic Modeling	UML Models can be reverse engineered out of a running system
Non-Functionals	Textual Annotations/Comments
Ambiguity	UML does not have a strict semantics. Stereotypes, Profiles can be used to reduce ambiguity
Accuracy	Basic Syntax (Well-formed) and OCL checking
Precision	UML Diagrams can become very detailed and be used to generate code
Viewpoints	Each kind of Diagram shows a different viewpoint
View Consistency	Little support offered

UML Overview

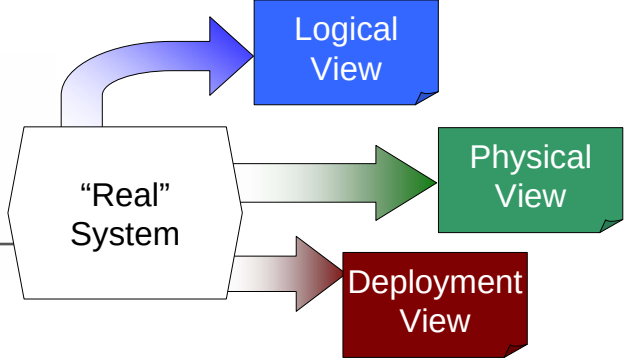
- Unified Modeling Language
- Visual notation for specifying, constructing and documenting structure and behavior of software systems
- Standardized and maintained by the OMG (Object Management Group)
- Current Version: 2.1.2 (November 2007)

- Object-oriented
- Multiple Views Support
- Target Programming Language Independent
- Unifies many existing Object-Oriented Analysis and Design notations:
  - Booch Diagrams
  - Rumbaugh's OMT
  - Jacobson's OOSE
  - Jacobson's Objectory
  - Harel's Statecharts
  - ...

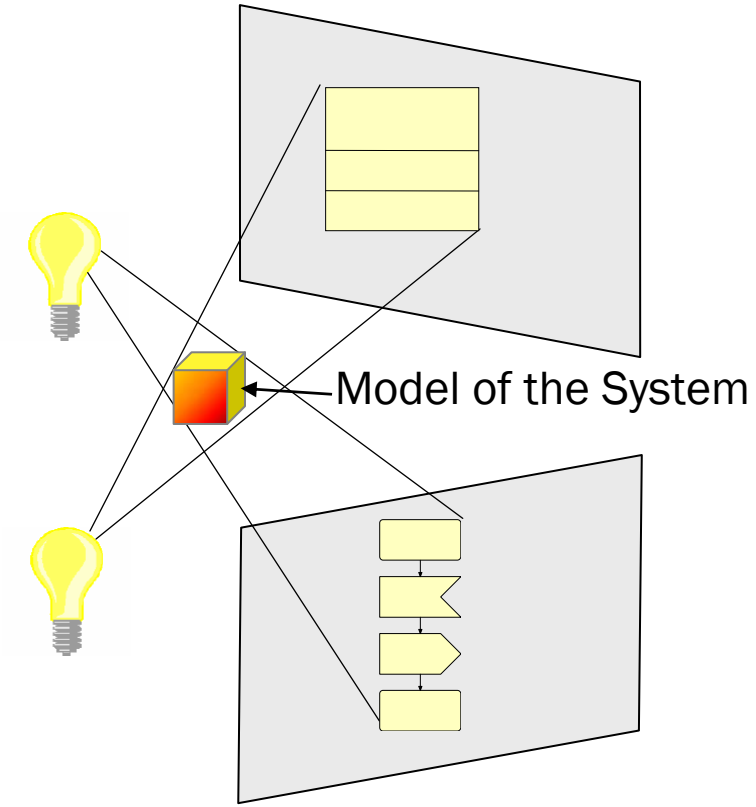
UML Goals

- Define an easy-to-learn and expressive, "general purpose" visual modeling language
- Unify existing modeling notations
- Incorporate industry best practices
- Address contemporary software development issues
  - scale, distribution, concurrency, model-driven engineering
- Provide flexibility for applying within different software engineering processes
- Enable model interchange between tools

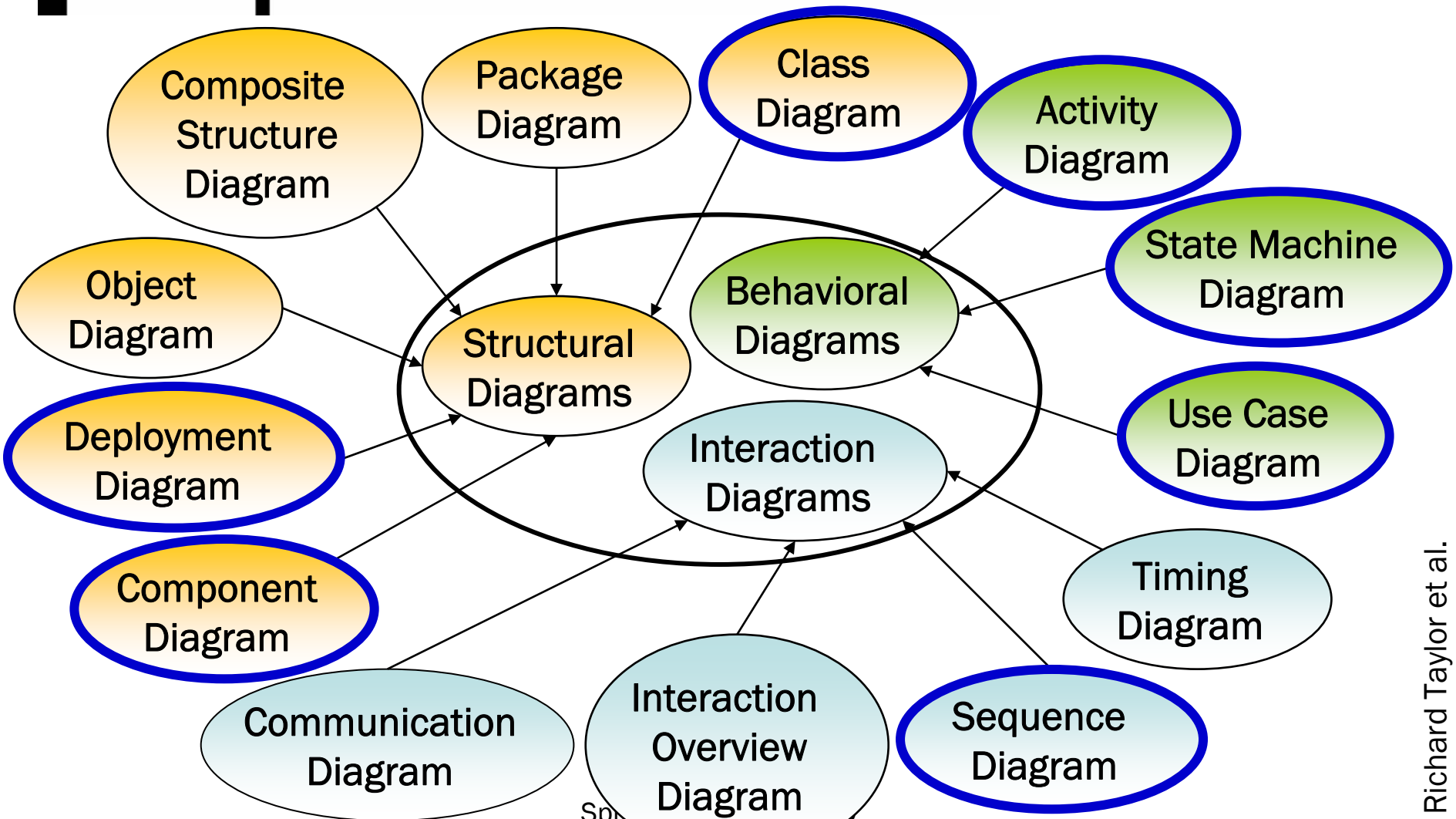
Multiple Views



- UML supports 13 different kinds of diagrams
- Each diagram groups different types of design decisions
- All diagrams together should give a "complete" view over the model of the system architecture



UML 2.0 Diagrams



From Richard Taylor et al.

Structural Diagrams

- **Object Diagrams** – show run-time relationships between object instances
- **Class Diagrams** – show the classes and their relationships (inheritance, aggregation, association)
- **Package Diagrams** – modularize the model description
- **Composite Structure** – define the internal structure of classes and components

- **Component Diagrams** – decompose the system into components, show their interfaces, dependencies and connections
- **Deployment Diagrams** – map components to the hardware execution environments

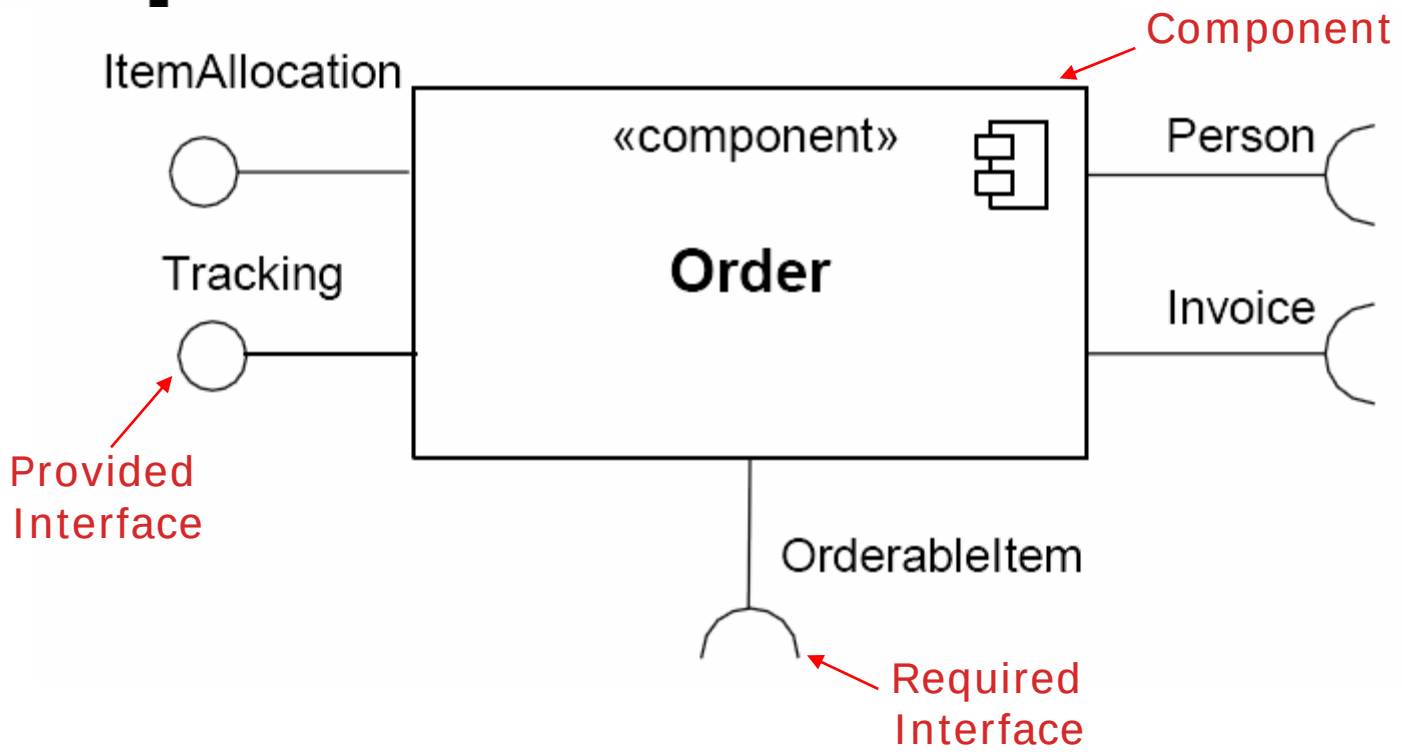
Behavioral Diagrams

- **Use Case Diagrams** – interaction scenarios between system and its environment/users
- **Activity Diagrams** – Flowcharts to model high level logic and business processes
- **State Machine Diagrams** – Model run-time states and the events that produce transitions between states

- **Communication Diagrams** – Show the interactions between a set of elements
- **Sequence Diagrams** – Show the interactions between a set of elements using swim-lanes
- **Interaction Overview Diagrams** – Activity+Sequence (connect together multiple diagrams)
- **Timing Diagrams** – State+Sequence viewed over time

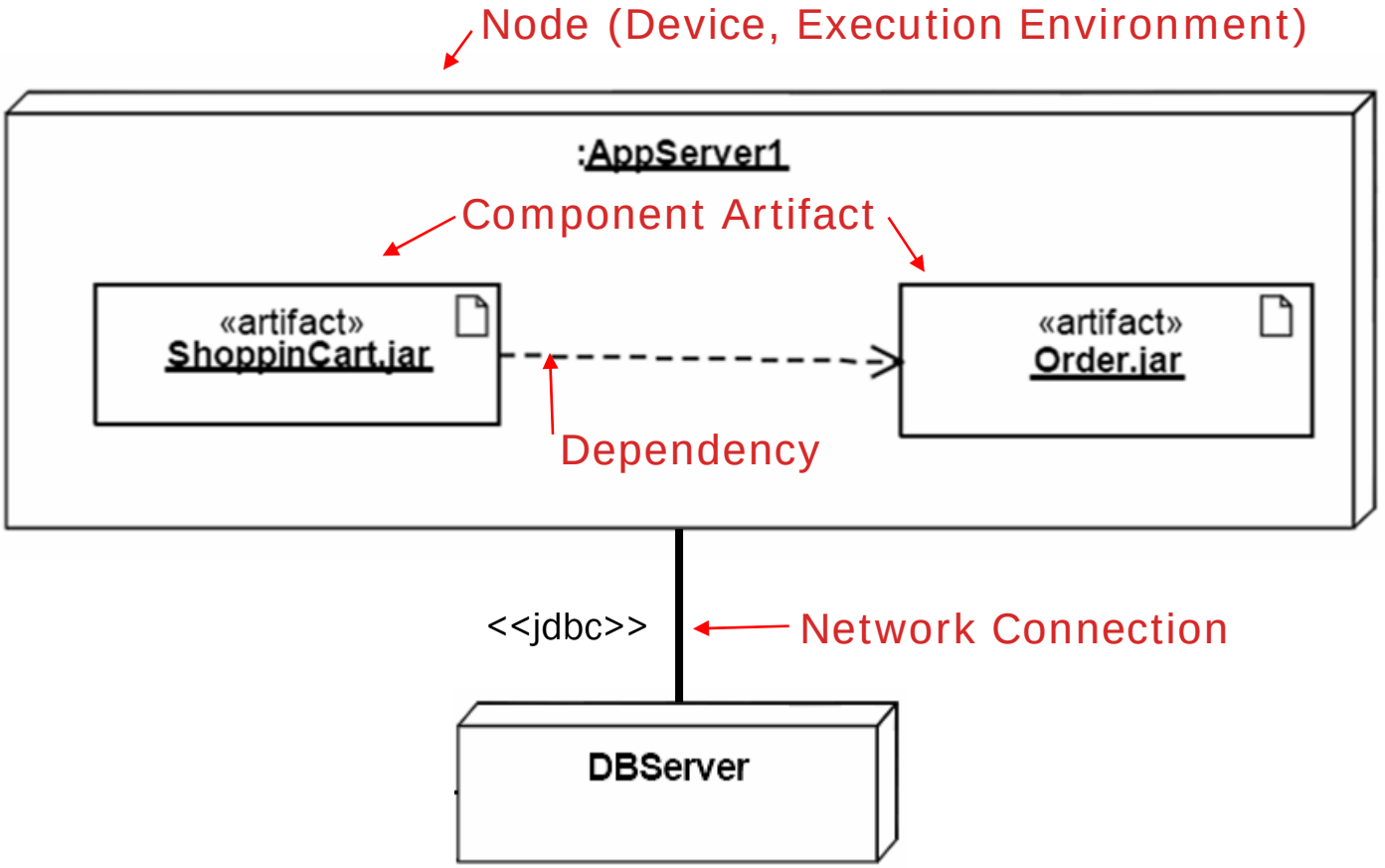
Component and Deployment Diagrams

Component Diagrams



- **Substitutability** = ability to transparently replace the content (implementation) of a component, provided its provided and required interface contracts are not modified

Deployment Diagrams



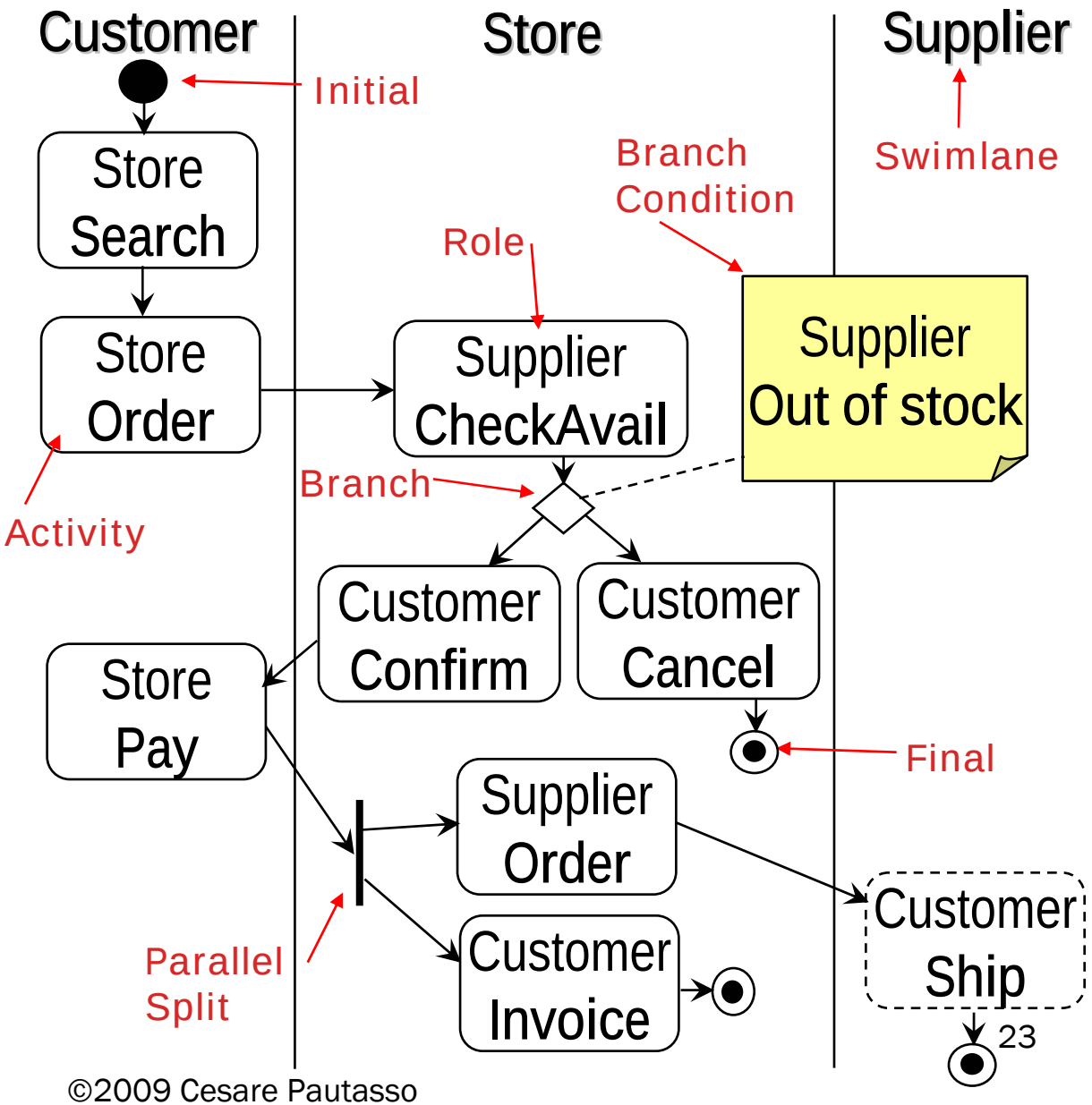
Deployment Diagrams

- Model the physical software architecture, including: the hardware, the software installed on it and the middleware providing the communication between the various **nodes** of the system
- Gives a static view of the run-time configuration of processing nodes and the components that run on them
- Trivial for monolithic, centralized systems

Activity and Sequence Diagrams

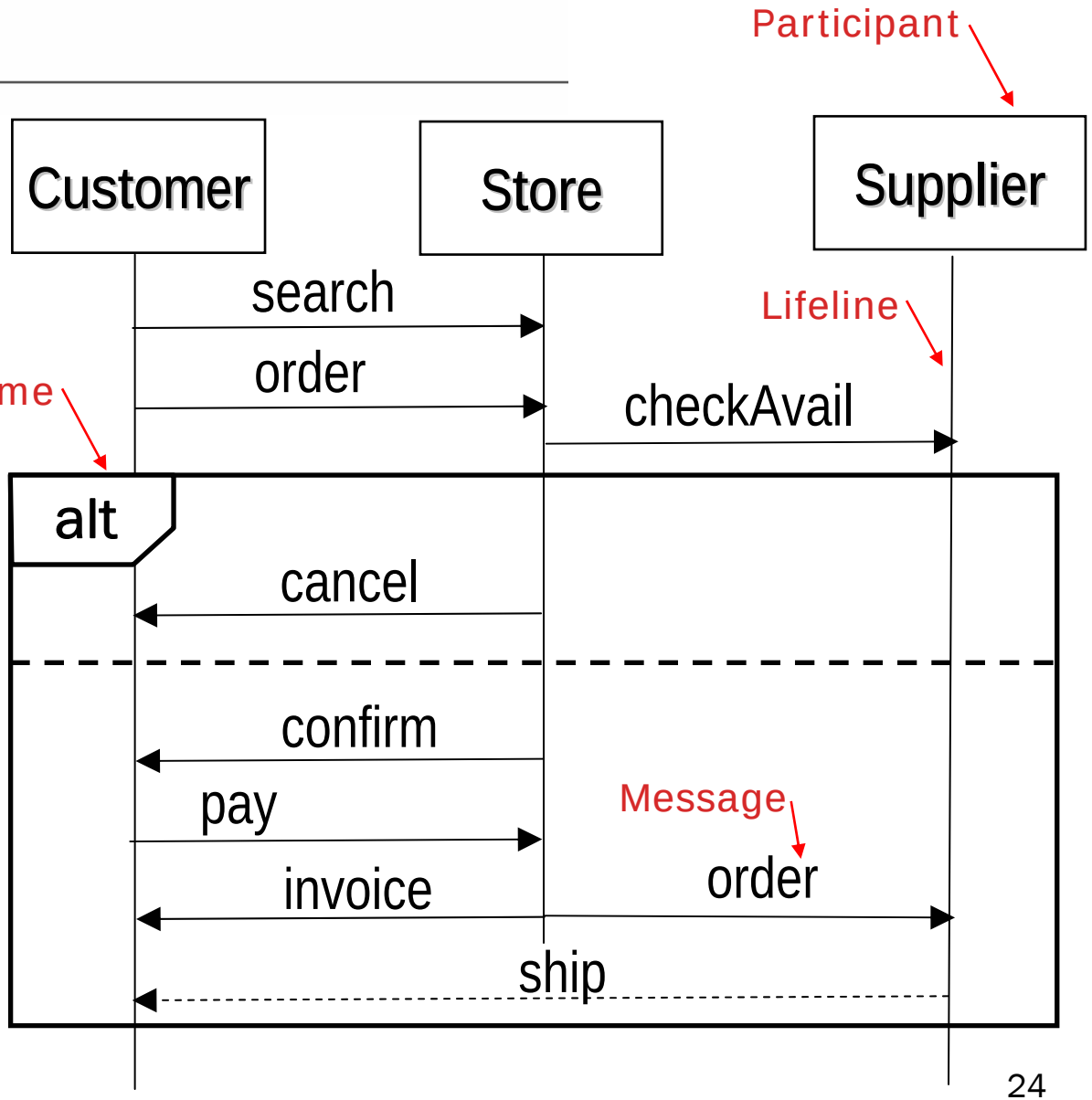
Activity Diagrams

- Model complex interactions within multiple components
- Model high-level business processes
- Concurrency semantics based on Petri-Nets



Sequence Diagrams

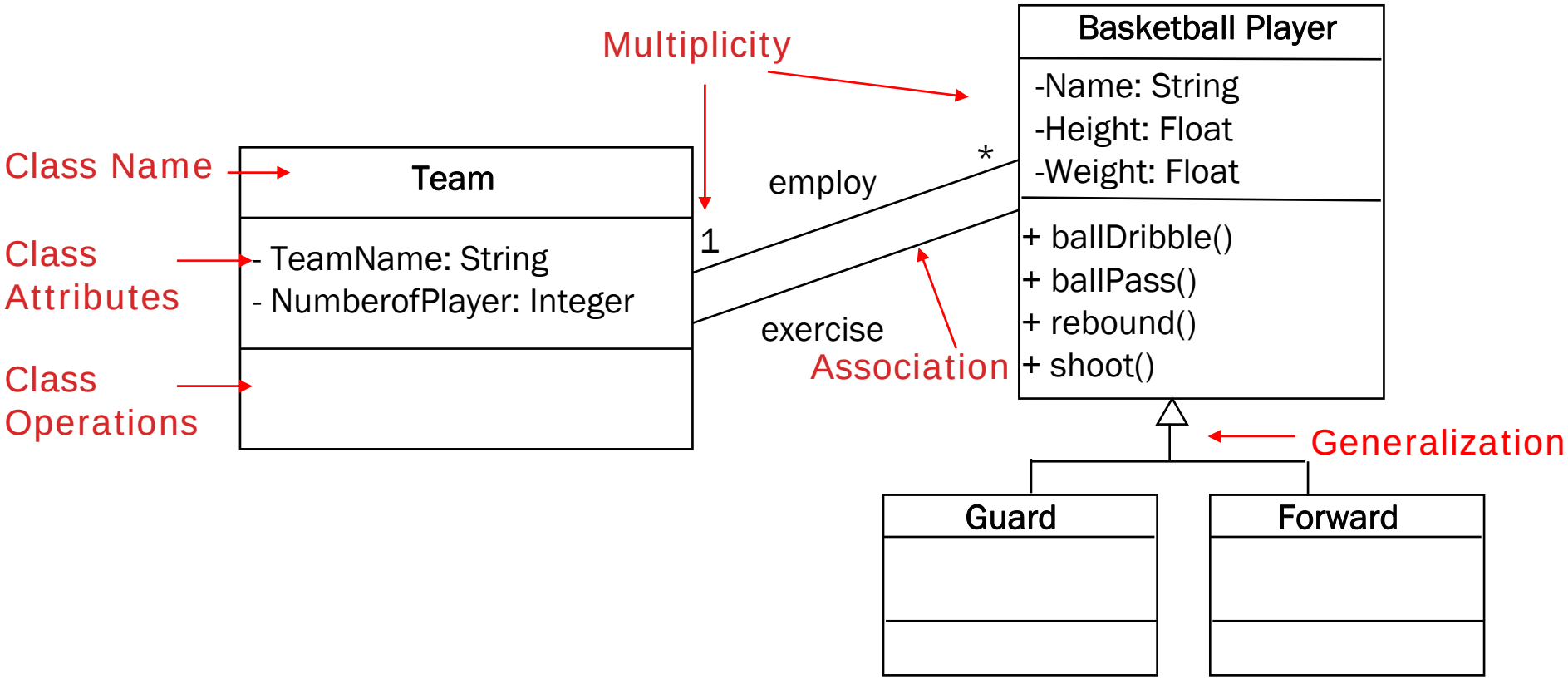
- Model concrete execution scenarios highlighting the lifelines of the participants
- Focus on interactions represented by ordered message exchanges.
- Frames are used to model **opt**ional **alt**ernative, **loop**, **ref**erence, **par**allel, invalid (**neg**), **critical**, interaction fragments



Class Diagrams

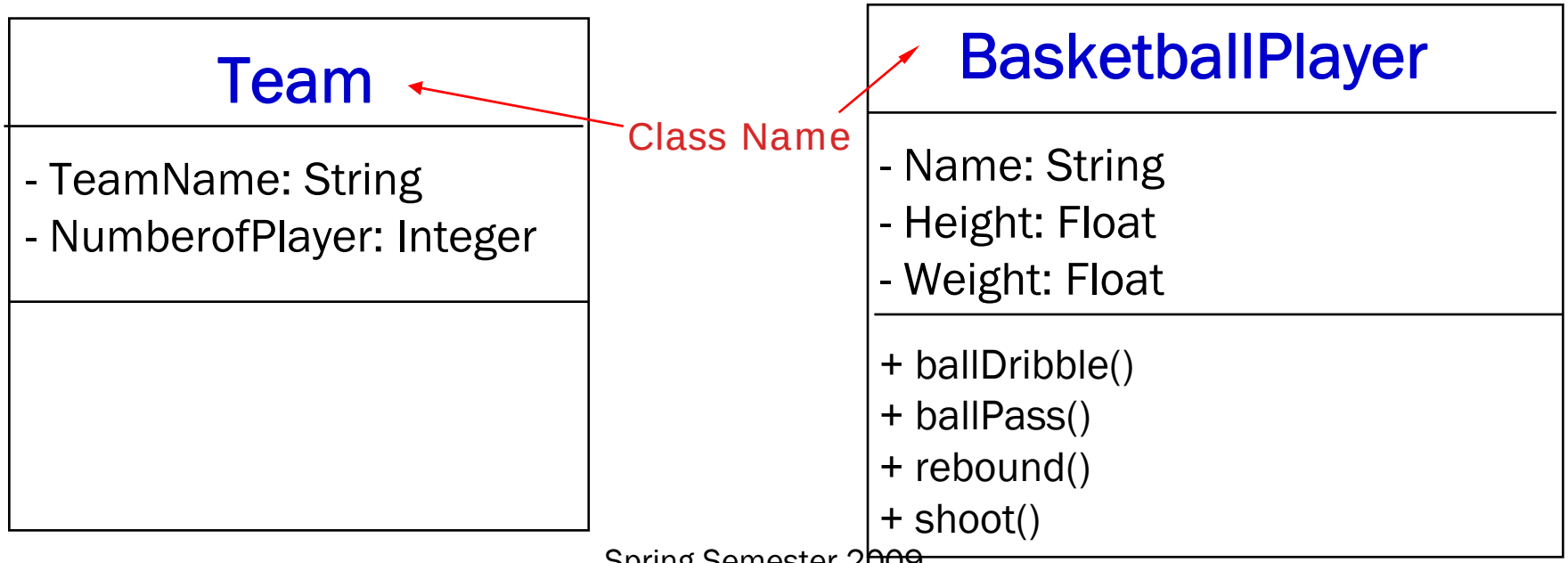
Class Diagrams

- Model the types of objects in the system and the relationships between them



Classes

- Most important building block of any object-oriented system
- Abstraction modeling the properties of a set of objects in the system
- Can be associated with other diagrams modeling their behavior



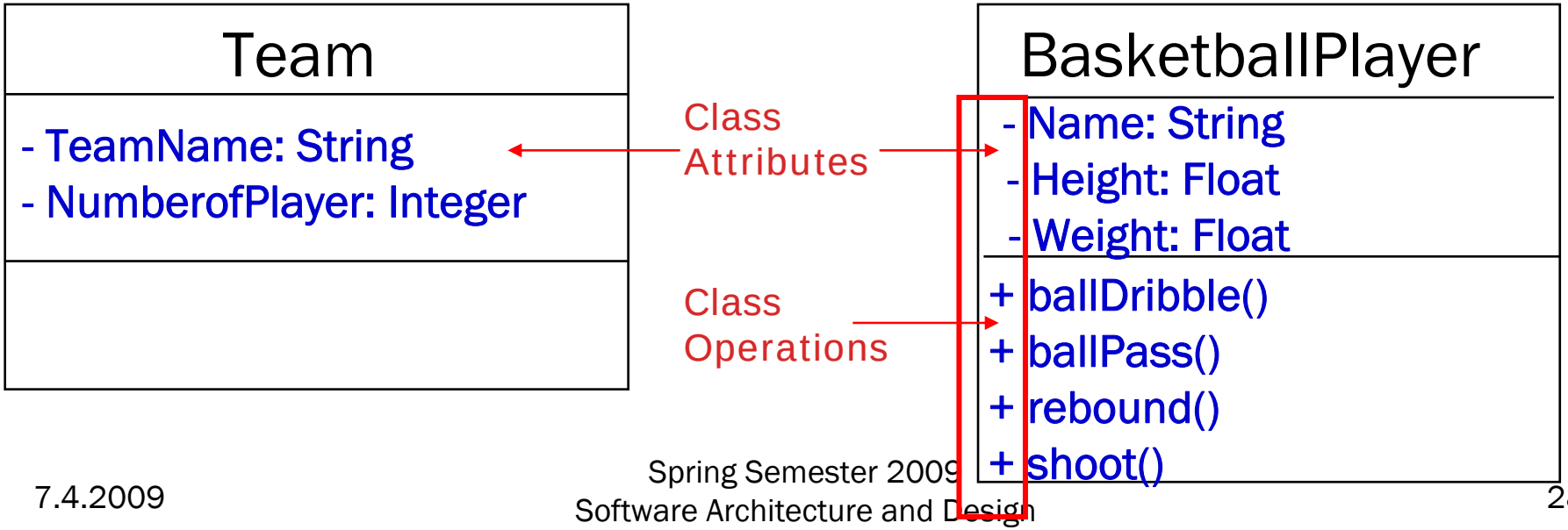
Attributes and Operations

- Attributes
  - Model properties of the class

  Syntax: `Name: Type`
- Operations
  - Model methods of the class

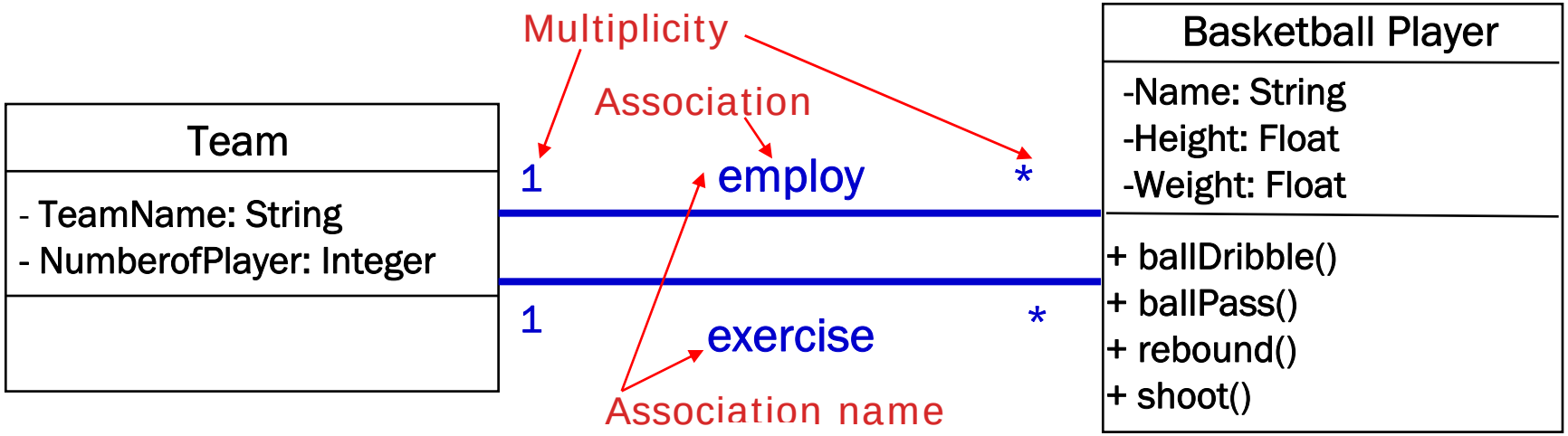
  Syntax: `Name(param1:type, param2:type, ...) : Result`

- Visibility:
  - \- Private
  - \+ Public
  - \# Protected
  - ~ Package



Association and Multiplicity

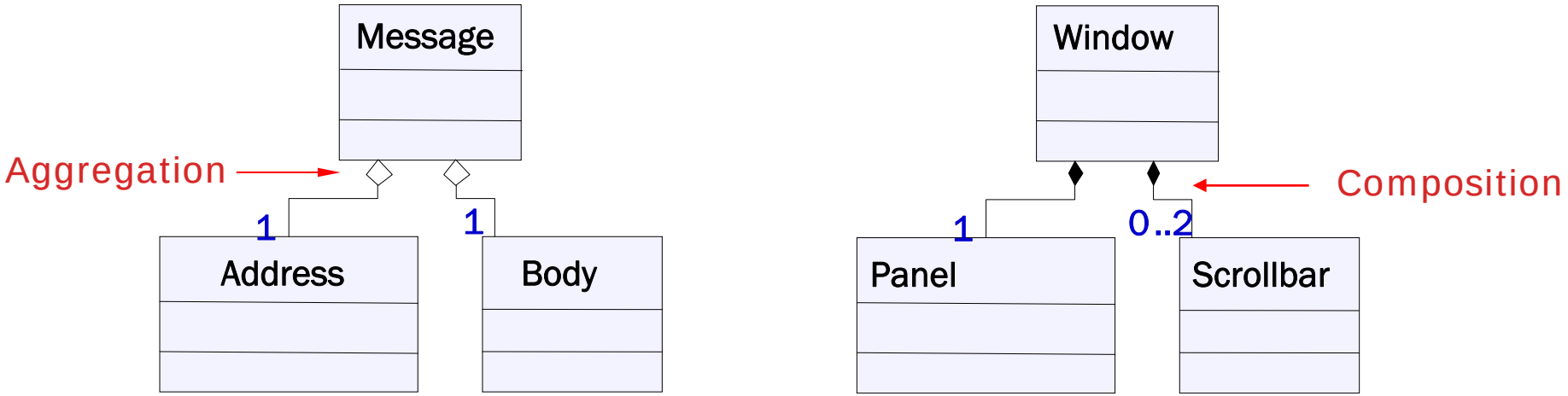
- Association
  - Relationship between classes that specifies the existence of a reference between their objects
- Multiplicity
  - Number of instances of one class related to ONE instance of the other class



"Team employs one or more basketball players"
"Many Basketball players exercise in one team"

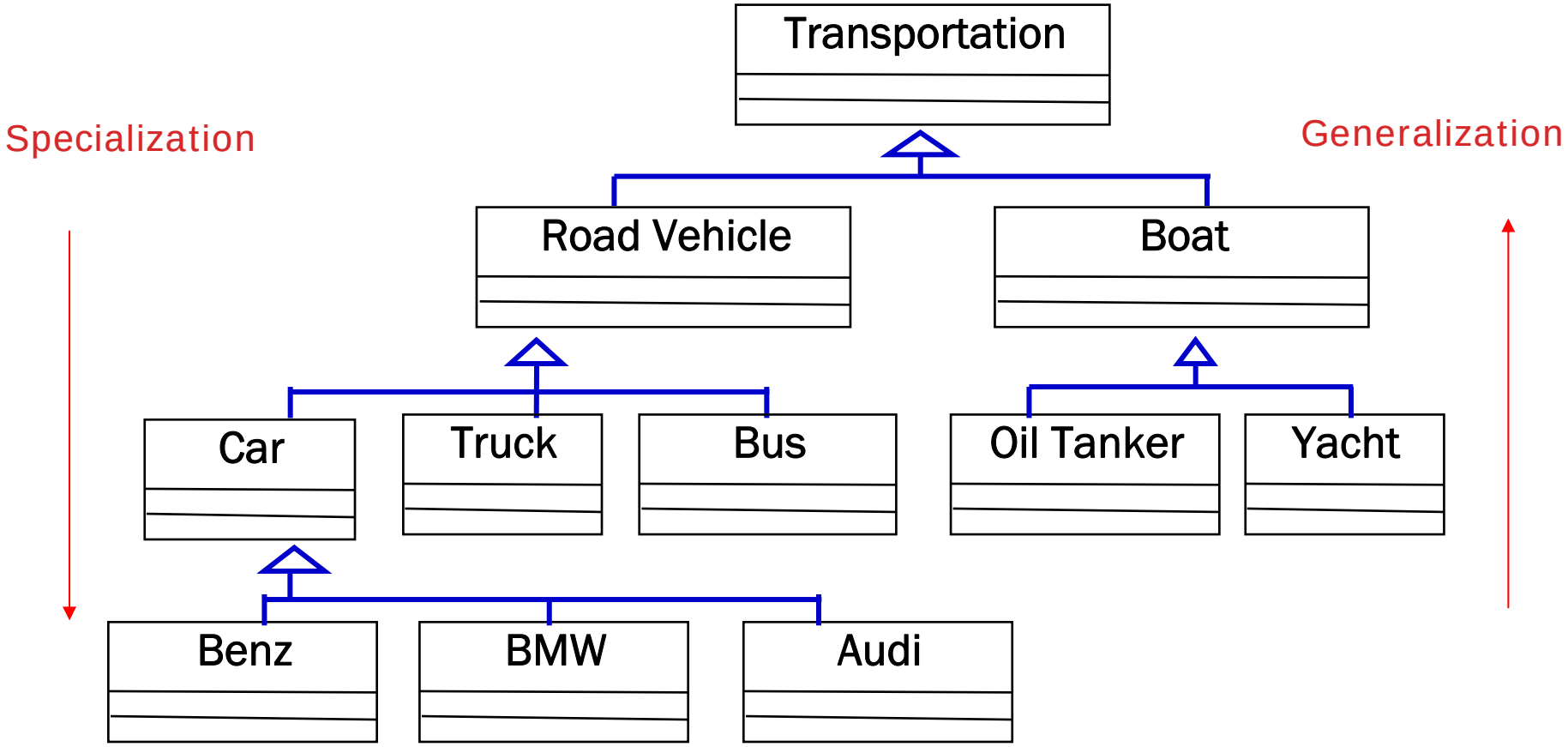
Aggregations and Compositions

- Aggregation
  - Weak "whole-part" relationship between elements
    - Address may exist without Message
    - Deleting Message does not delete Address
- Composition
  - Strong "whole-part" relationship between elements
    - Window 'contains a' Scrollbar



Inheritance

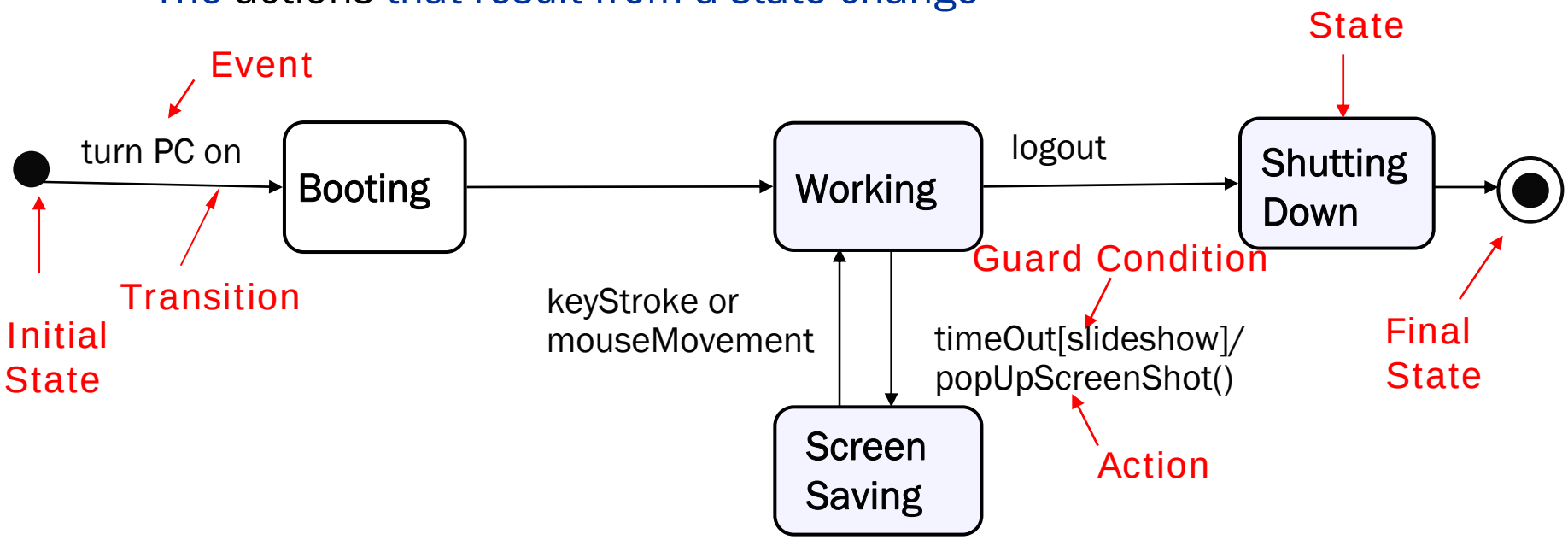
- Abstract commonalities of subclasses in their superclass
  - All attributes and operations of the superclass are also in the subclasses



State Machine Diagrams

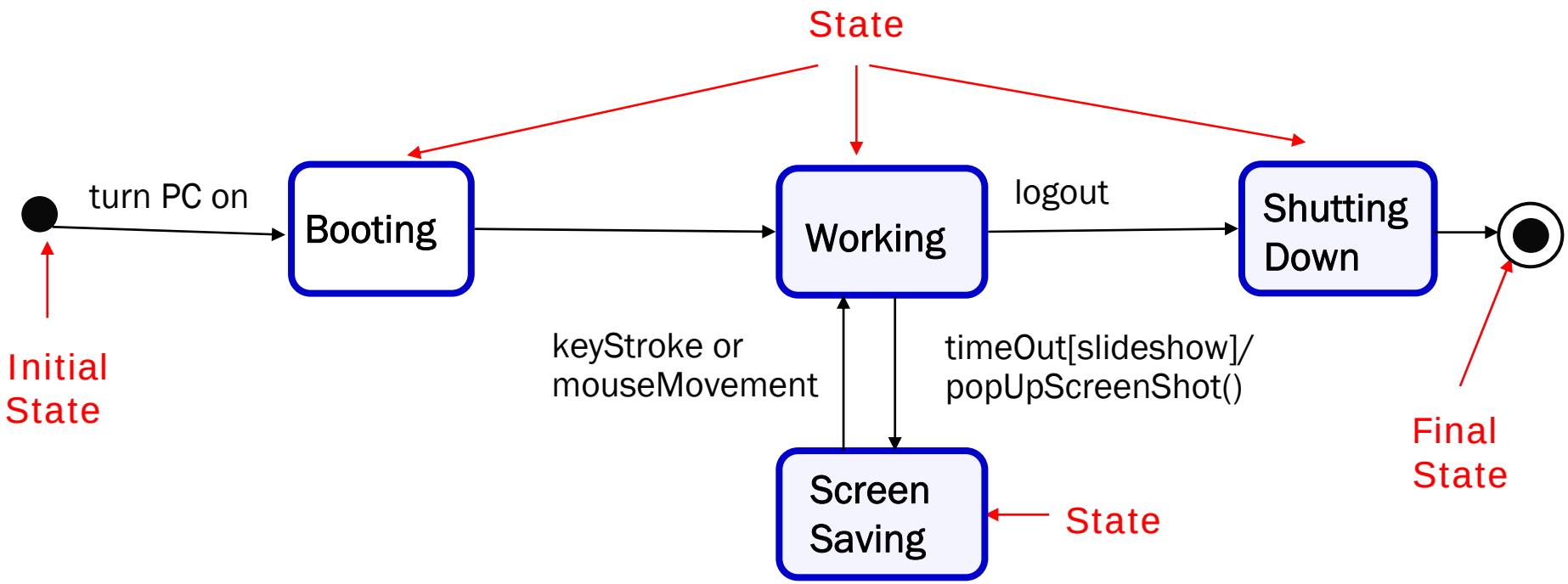
State Machine Diagrams

- Describe the dynamic behavior of objects over time by modeling their lifecycle showing:
  - The event that cause (or trigger) a transition between states
  - The conditions that guard the transition
  - The actions that result from a state change



States

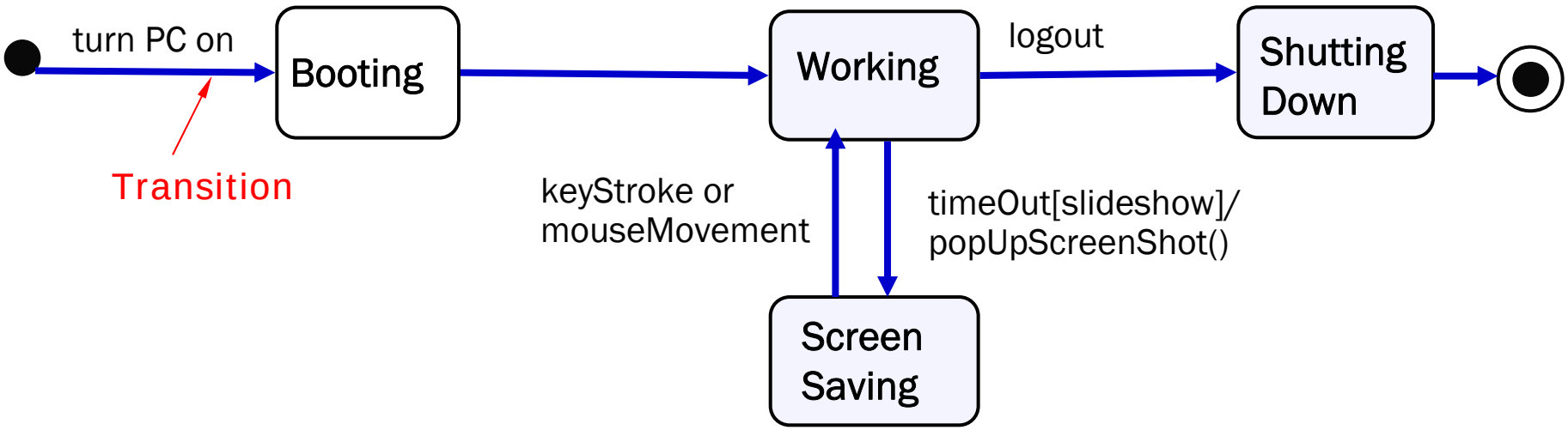
- Situation during the lifecycle of an object
  - Models some condition, some activity to be performed or waits for some event



Transition

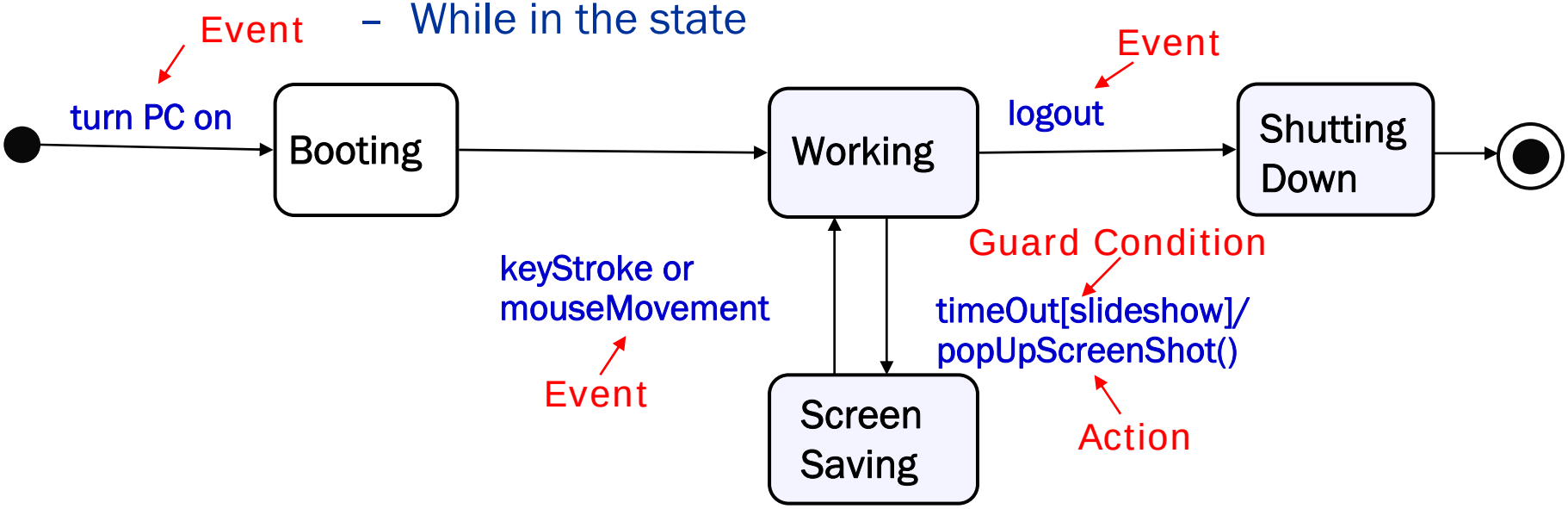
Syntax: `event(arguments)[condition]/action`

- Change state (may be triggered by an event)
- Occur only when guard condition is true



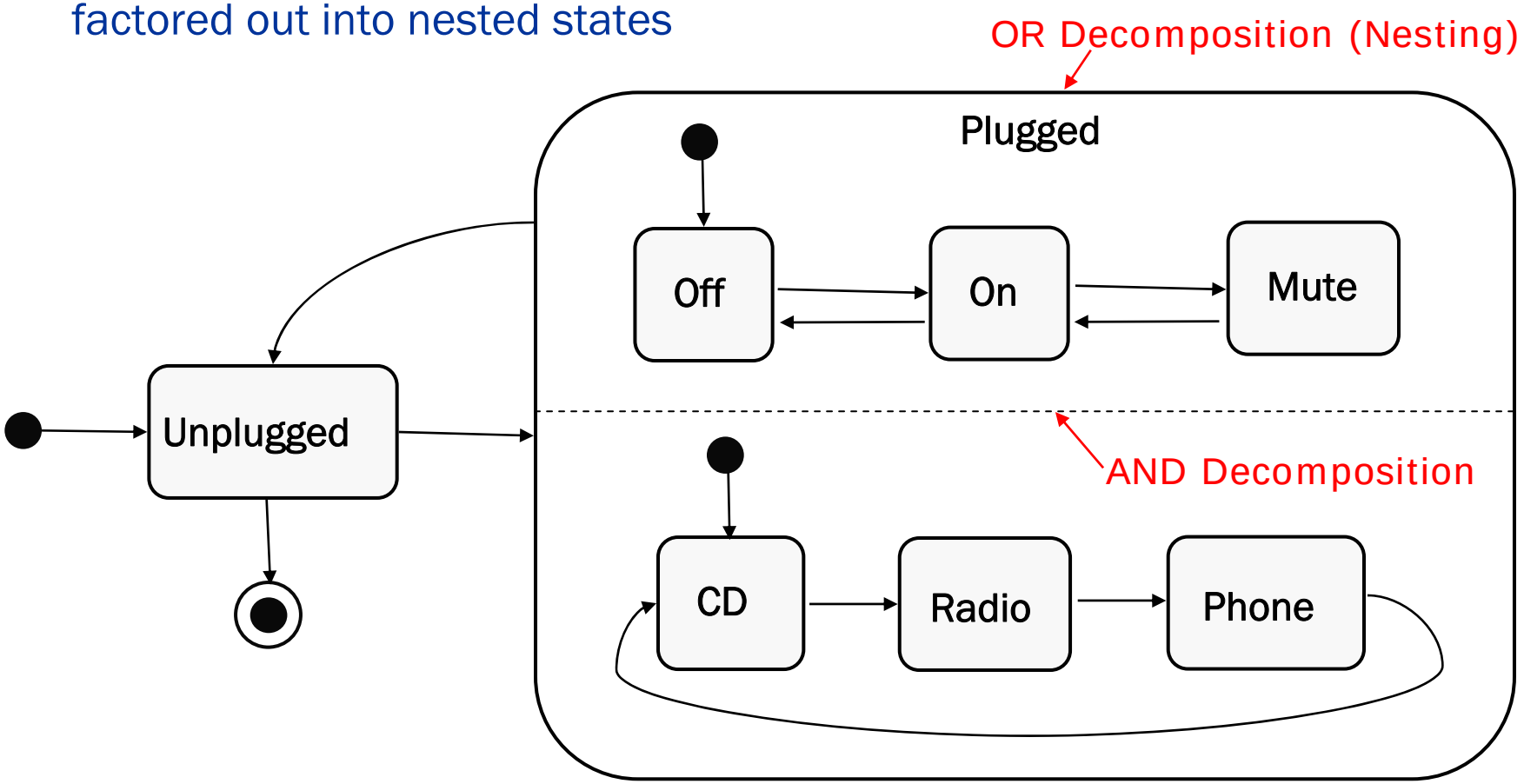
Event and Action

- Event (Trigger the state change)
- Action (Call to operation of the object)
  - During the transition
  - Before entering a state
  - After exiting a state
  - While in the state



Nested States

- To avoid the combinatorial explosion of the model, common states can be factored out into nested states



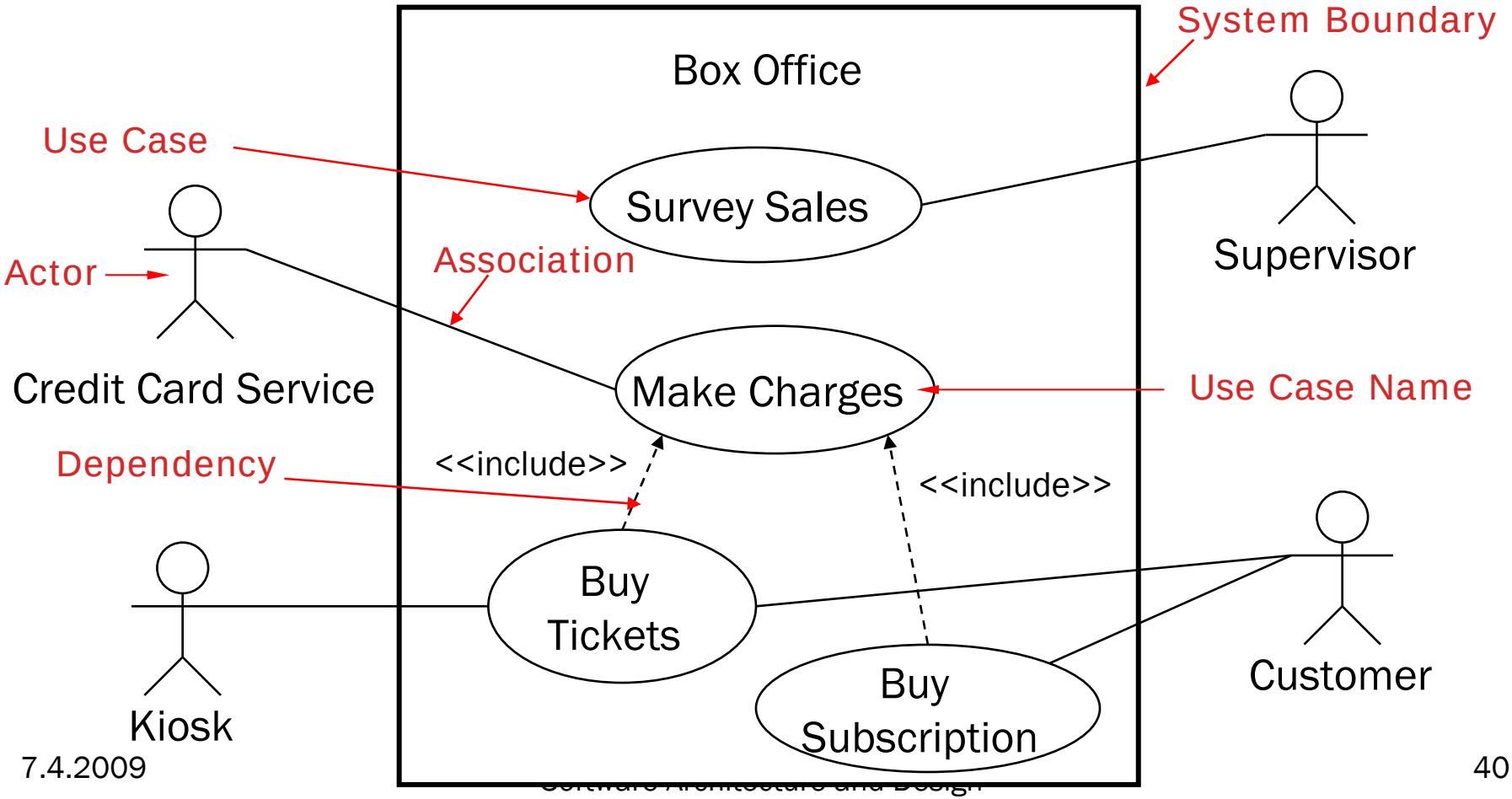
Use Case Diagrams

What is a Use Case?

Use Case ~ An observable behavior or coherent set of behaviors triggered by events sent to the system from actors found in the outside environment: human users, other systems, timers, or hardware components.

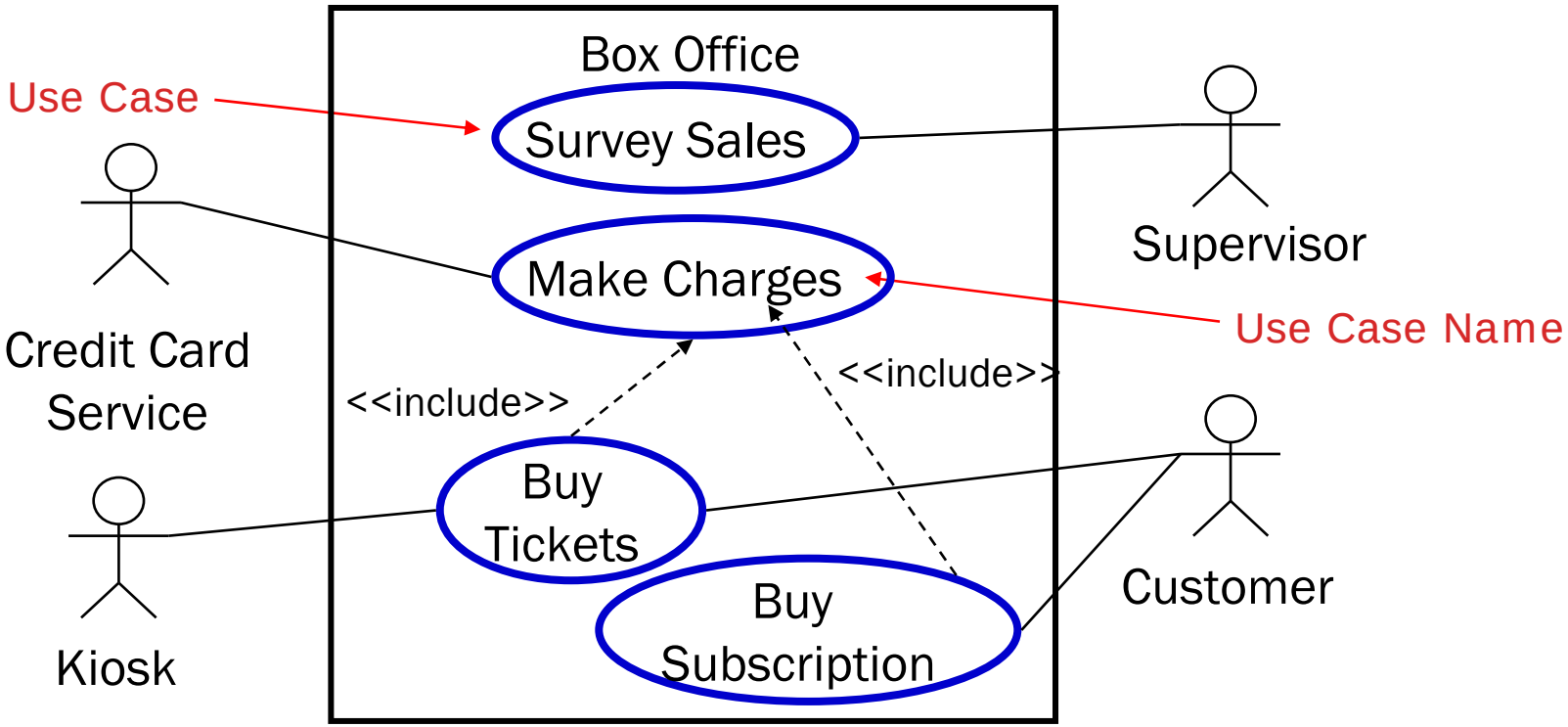
Use Case Diagrams

- Describe WHAT the system will do at a high-level



Use Case

- Unit of functionality (or feature) expressed as the interaction among actors and the system

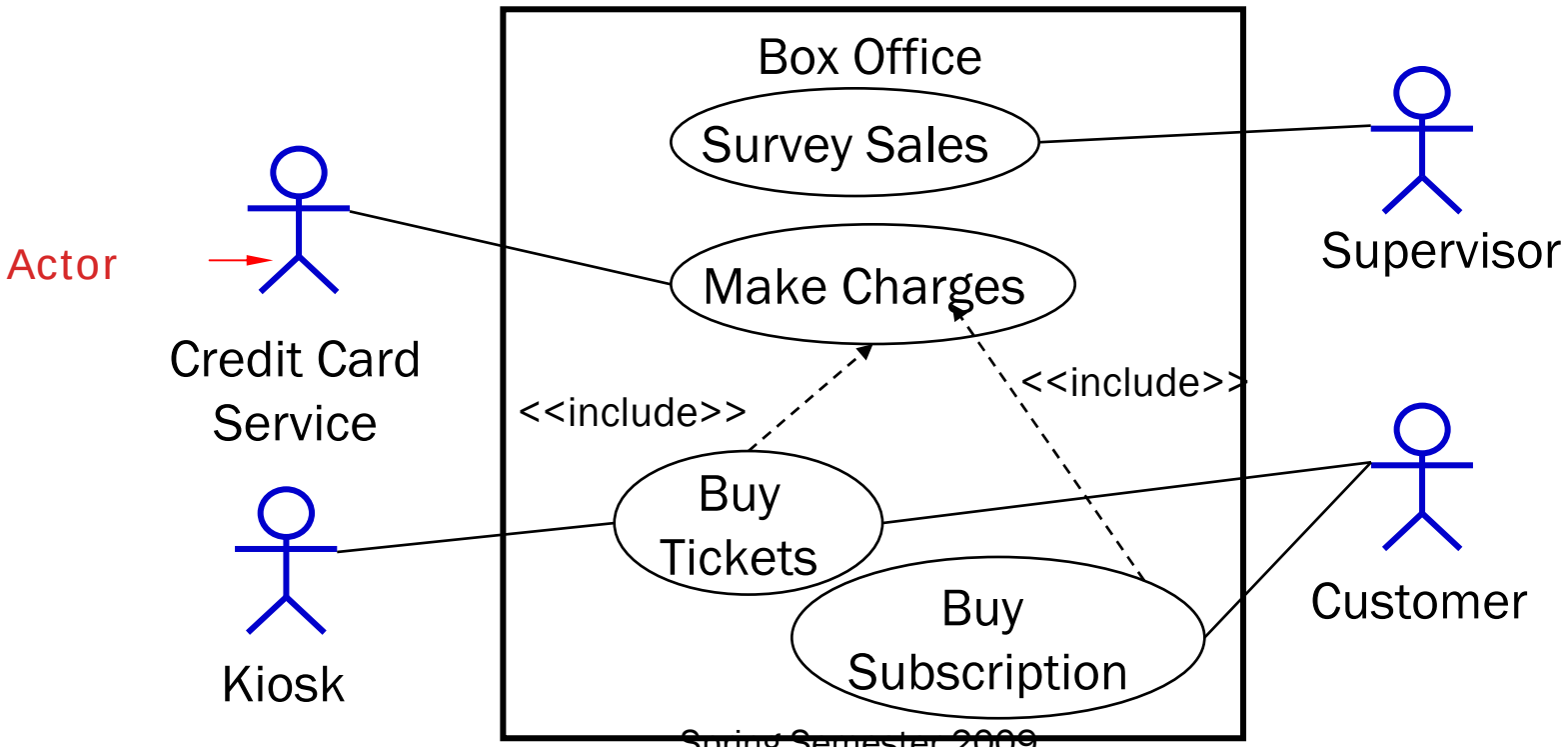


Identifying Use Cases

- Which functions does the actor require from system?
- Does the actor need to read, create, destroy, modify, or store some kind of information in the system?
- Does the actor have to be notified about events from the system?
- Could the actor's daily work be simplified or made more efficient through a specific feature of the system?

Actor

- Entities that interact with the system:
  - User roles, external systems, devices

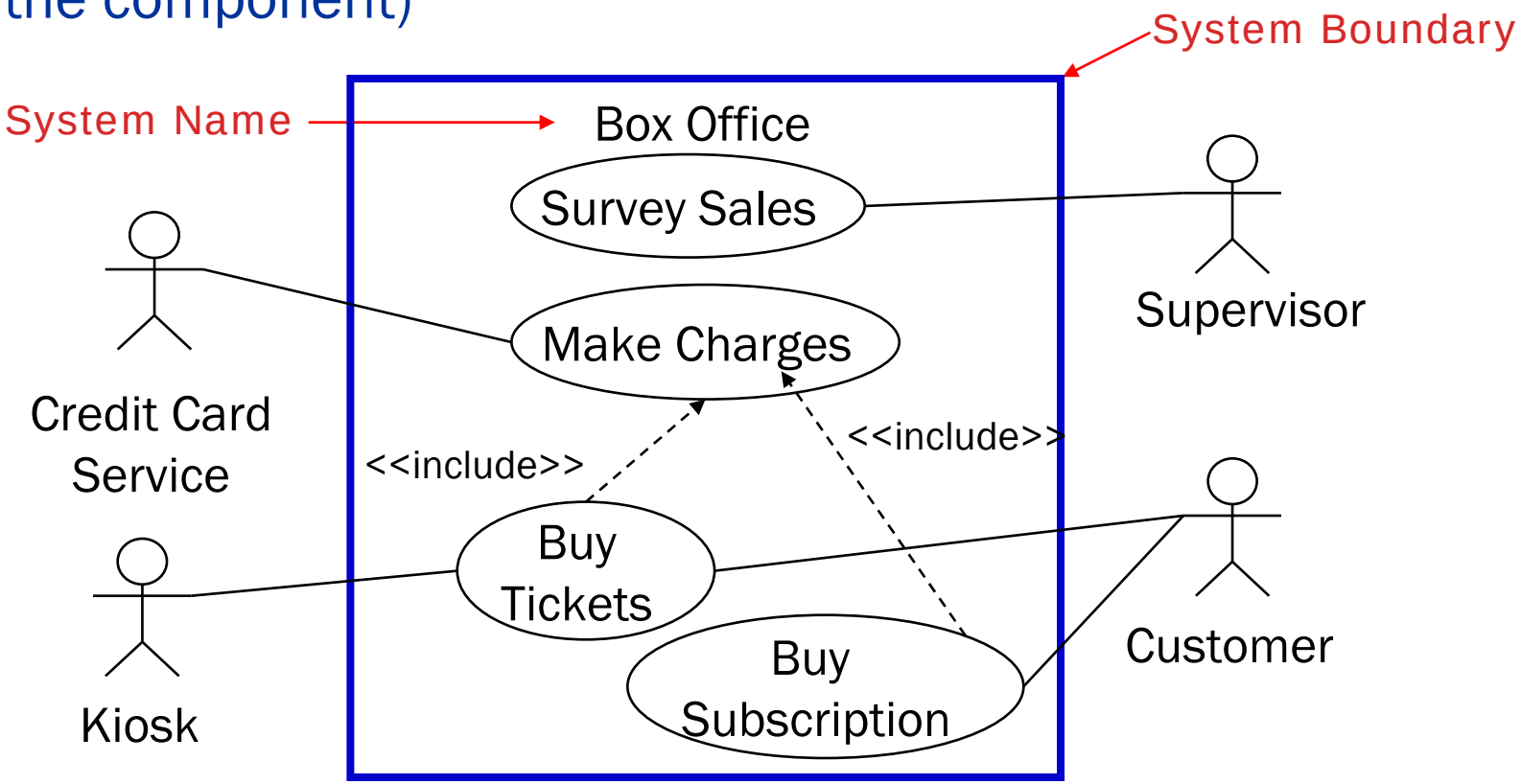


Identifying Actors

- Primary Actors:
  - Who will use the main functionality of the system?
  - Who will need support from the system to their daily tasks?
  - Who or what has an interest in the results (the value) that the system produces?
- Secondary Actors:
  - Who will need to maintain, administrate, and keep the system working?
  - With which other systems does the system need to interact/depend on?

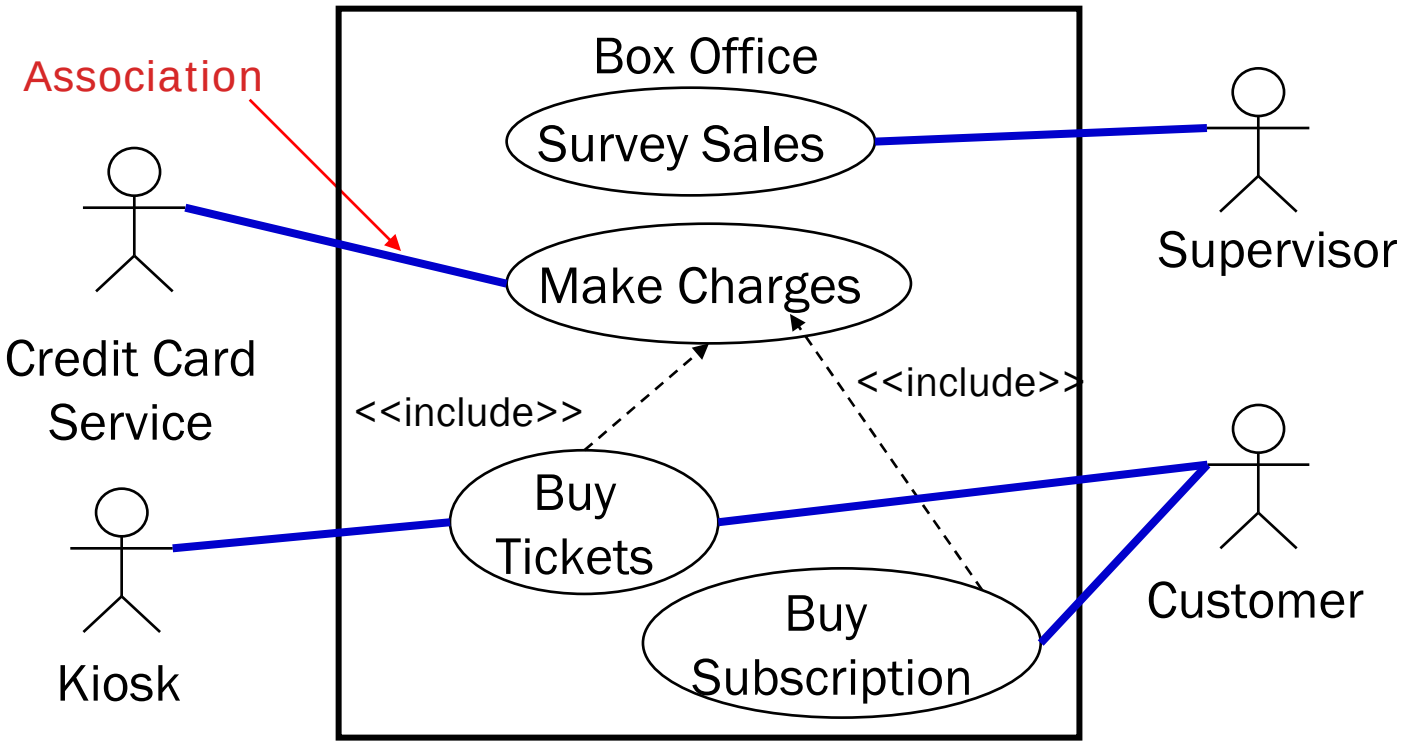
System Boundary

- Group the use cases implemented by the system (or the component)



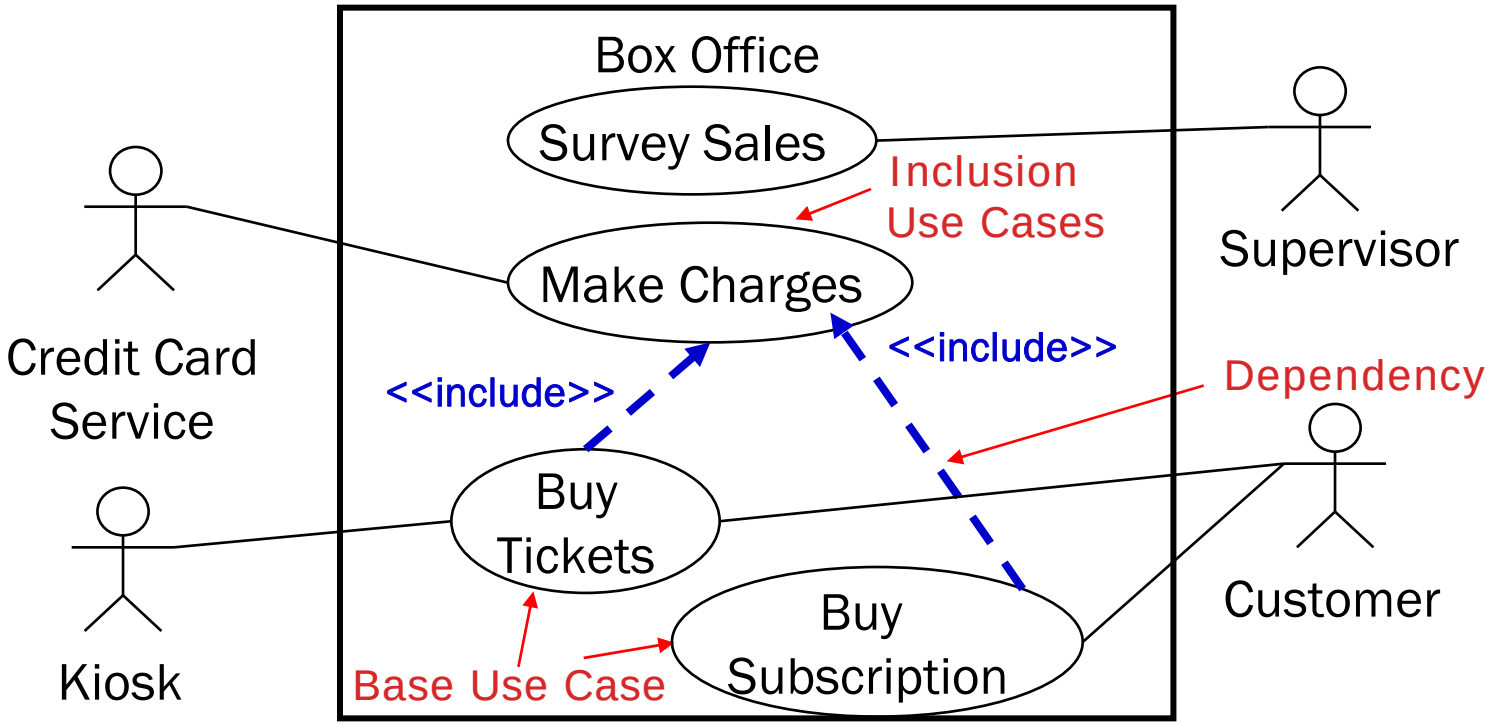
Association

- Model bi-directional communication between the actor and the system within a specific use case



Dependency – Include

- Reuse behavior common to several Use Cases
- Helps to decompose the system reusing components



Use Case Diagrams Summary

- The diagram itself just shows an overview listing the use cases supported by the systems and the actors involved in them
- The use cases themselves are usually described using text and other more precise interaction diagrams
- Use case diagrams are often used to model requirements, plan the development of the project, assign priorities, and communicate with customers.

References

- A. Cockburn, **Writing Effective Use Cases**, Addison-Wesley, October 2000
- S. W. Ambler, **The Object Primer: Agile Model Driven Development with UML 2**, 3rd Ed, Cambridge University Press, 2004
- D. Rosenberg, K.Scott **Use Case Driven Object Modeling with UML : A Practical Approach**, Addison-Wesley, 1999
- I. Jacobson, **Object-Oriented Software Engineering: A Use Case Driven Approach**, Addison-Wesley, 1999
- David Harel and Eran Gery, "Executable Object Modeling with Statecharts", *IEEE Computer*, July 1997, pp. 31-42.
- http://www.omg.org/spec/UML/2.1.2/
  - Infrastructure (224 pages)
  - Superstructure (768 pages)

Università
della
Svizzera
italiana

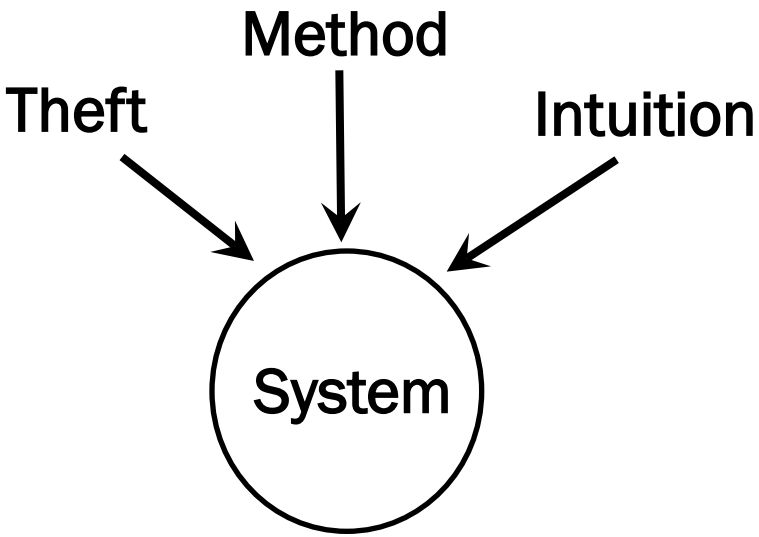
Facoltà
di scienze
informatiche

More Architectural Patterns

Prof. Cesare Pautasso

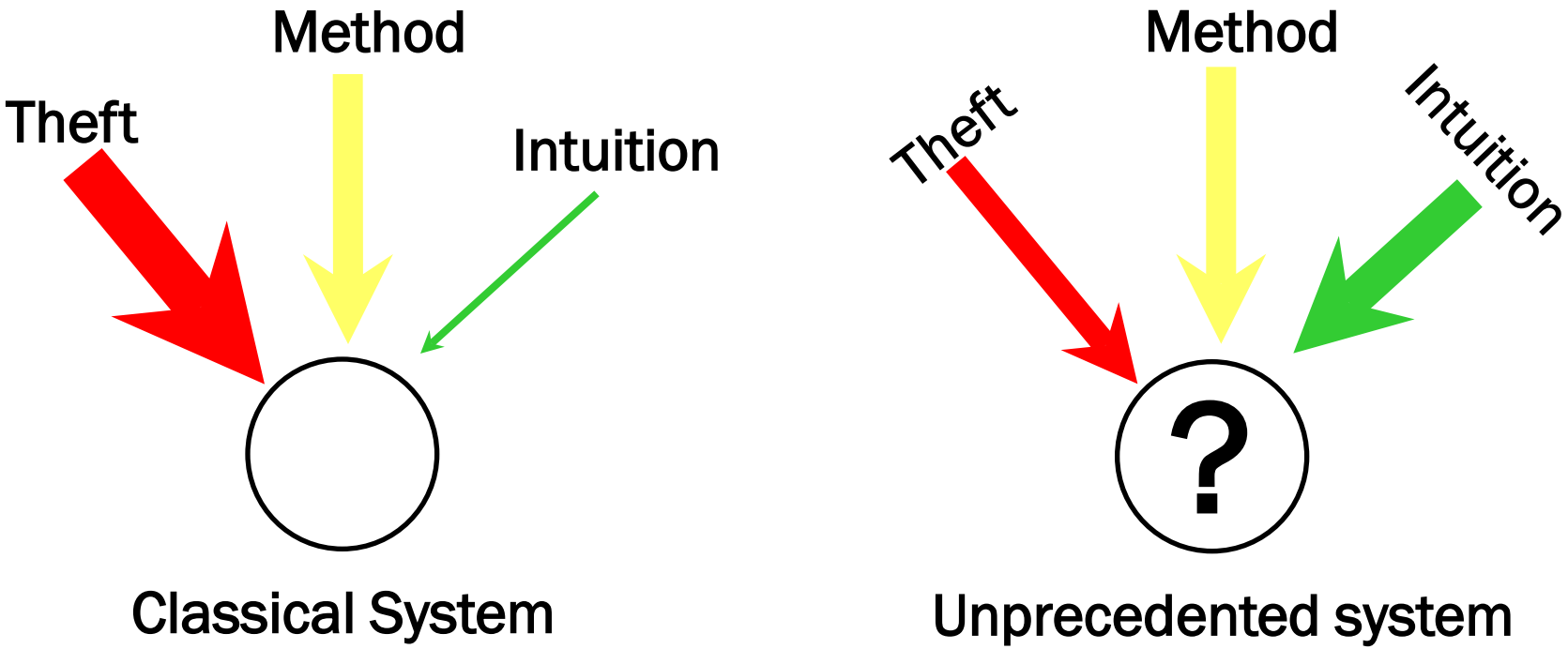
http://www.pautasso.info

How to Design



From Grady Booch

How to Design



From Grady Booch

Why Patterns?

Patterns help you build on the collective experience of many skilled software engineers

Architectural Patterns

- Goal:
  practical advice and solutions to concrete design problems
- Definition:

> A pattern is a standard solution
> to common and recurring design problems

- Related:
  related (but different) to Object-Oriented Design Patterns
- Hint:
  these patterns should be applied where they actually solve a problem
- Warning:
  as with all design decisions, all consequences should be evaluated and implications fully understood

Styles vs. Patterns

From R. Taylor

Patterns

- Layered
  - State-Logic-Display
  - Model-View-Controller
- Service Oriented Architecture
  - Interoperability
  - Service Directory
- Notification
  - Event Monitor
  - Observer
  - Publish/Subscribe
  - Messaging Bridge
- Composition
  - Scatter/Gather
  - Master/Slave
  - Synchronous
  - Asynchronous
- Testing
  - Faux Implementation
  - Wire Tap
- Optimization
  - Data Transfer Object
  - Partial Population

Layered

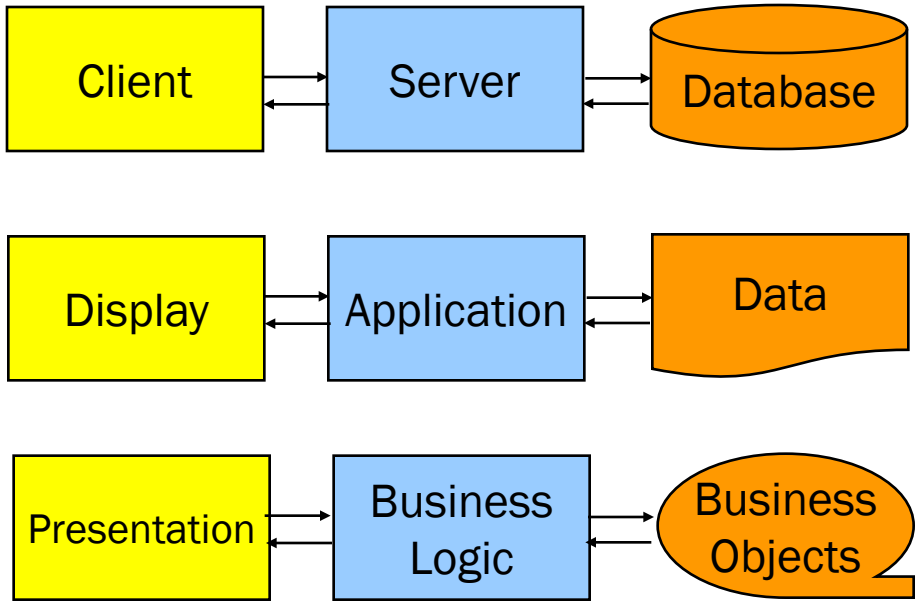
State-Logic-Display

Model-View-Controller

State-Logic-Display

- Goal:
  separate elements with different rate of change
- Solution:
  cluster elements that change at the same rate together
- Pattern:
  apply the layered style to separate the user interface (display/client), from the business logic (server), from the persistent state (database) of the system

State-Logic-Display



Client is any user or program that wants to perform an operation with the system

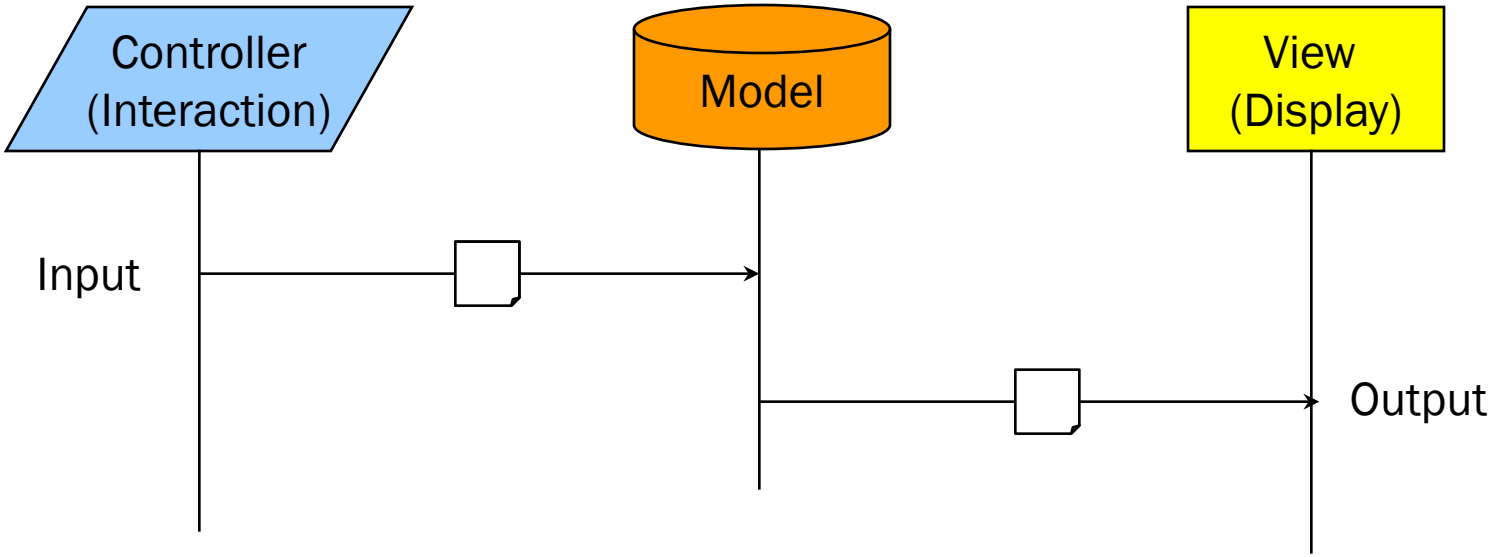
The application logic determines what the system actually does (code, rules, constraints)

The state layer deals with the organization (storage, indexing, and retrieval) of the data used by the application logic

Model-View-Controller

- Goal:
  support different means of interaction and display of the same content
- Solution:
  separate content (model) from presentation (output) and interaction (input)
- Pattern:
  model objects are completely ignorant of the UI. When the model changes, the views react. The controller's job is to take the user's input and figure out what to do with it. Controller and view should (mostly) not communicate directly but through the model.

Model-View-Controller



User Input is handled by the Controller which may change the state of the Model

The View displays the current state of the Model (and may receive notifications about state changes by observing it)

Università
della
Svizzera
italiana

**Facoltà
di scienze
informatiche**

Service Oriented Architecture

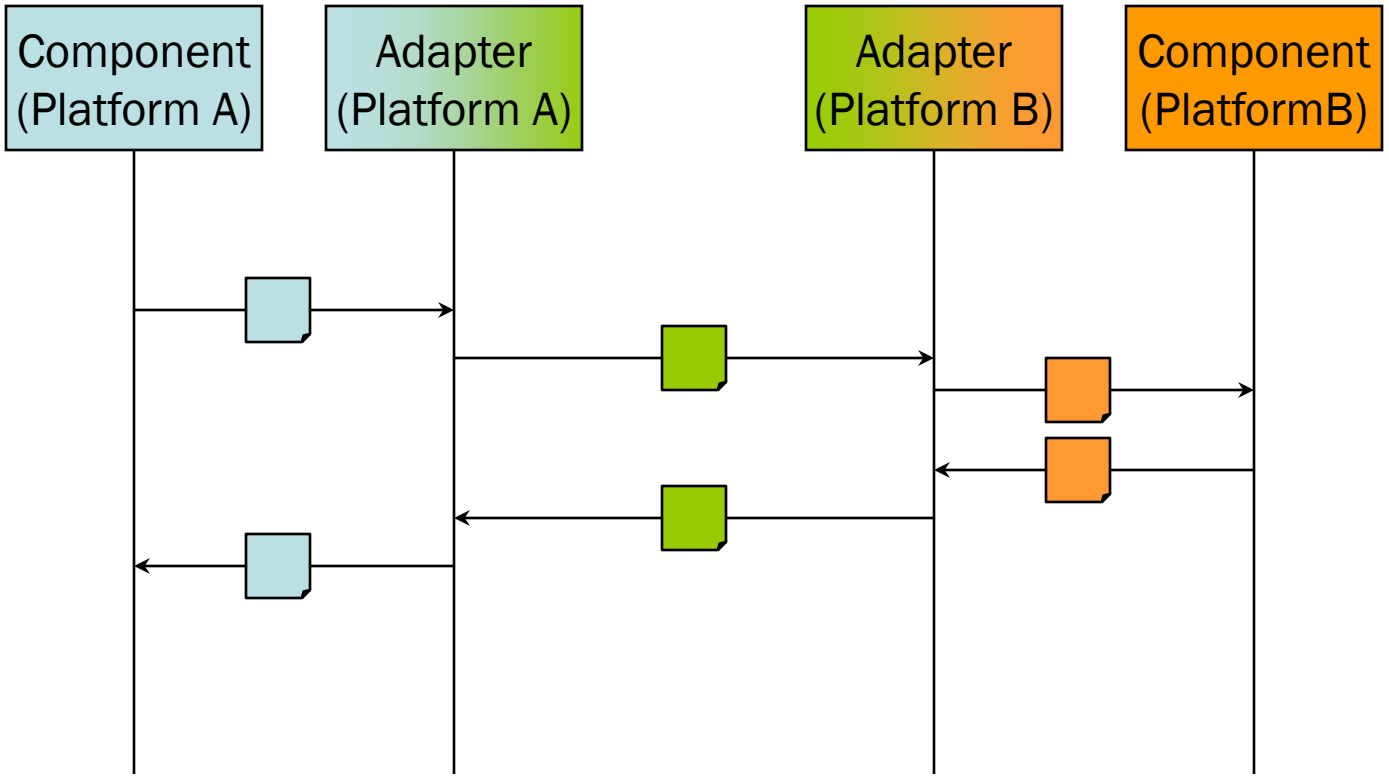
Interoperability

Service Directory

Interoperability

- Goal:
  enable communication between different platforms
- Solution:
  map to a standardized intermediate representation and communication style
- Pattern:
  components of different architectural styles running on different platforms are integrated by wrapping them with adapters. These adapters enable interoperability as they implement a mapping to common means of interaction

Interoperability Pattern



Components of different platforms can interoperate through adapters mapping the internal message format to a common representation and interaction style

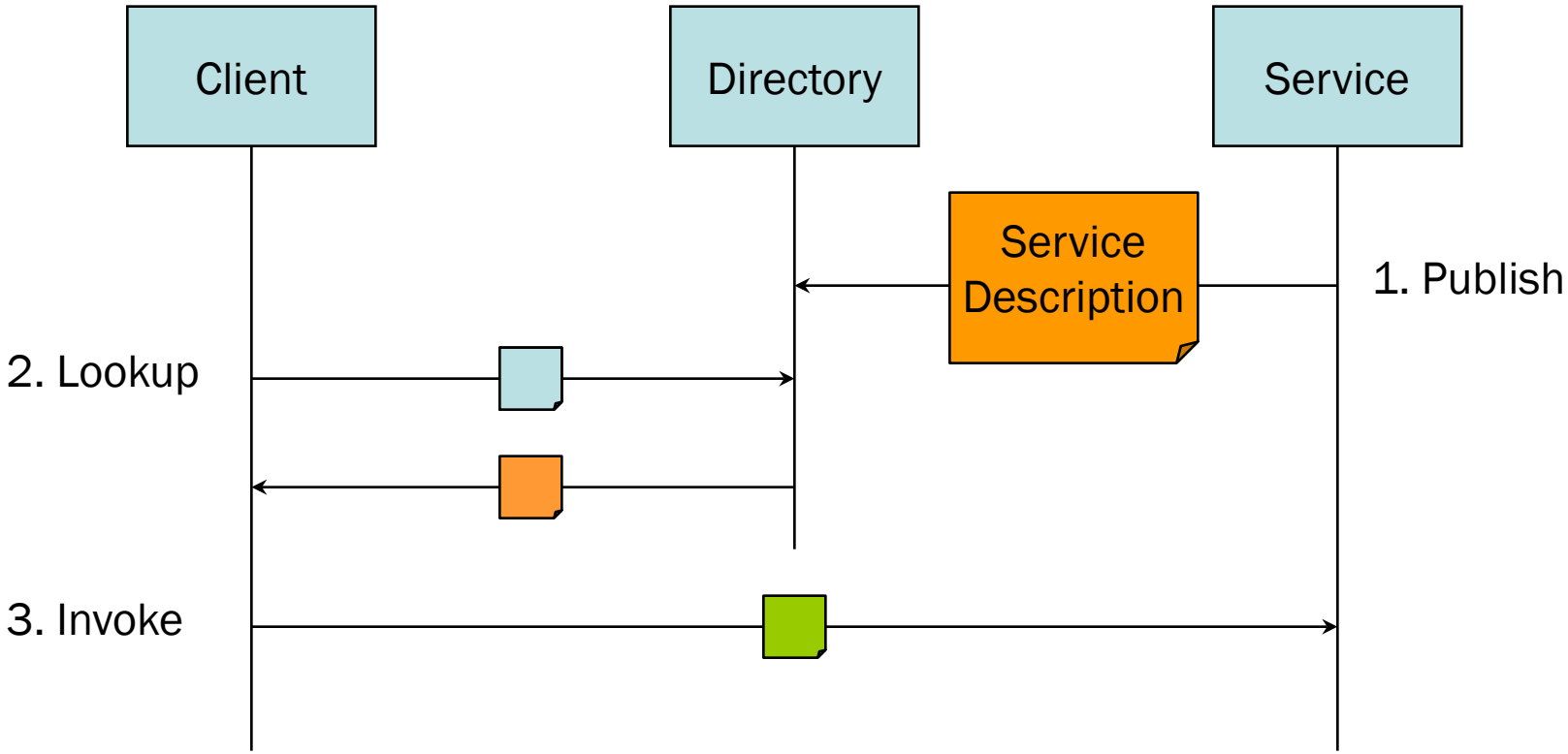
Interoperability: Hints

- Why two adapters?
  - For Point to Point solutions one adapter is enough (direct mapping)
  - In general, when integrating among N different platforms, the total number of adapters is reduced by using an intermediate representation
- Adapters do not have to be re-implemented for each component, but can be generalized and shared among all components of a given platform
- This pattern helps to isolate the complexity of having to deal with a different platform when building each component. The adapters make the heterogeneity transparent.
- Warning: performance may suffer due to the overhead of applying potentially complex transformations at each adapter

Service Directory

- Goal:
  facilitate location transparency
  (avoid hard-coding service addresses in clients)
- Solution:
  use a directory service to find service endpoints based on abstract descriptions
- Pattern:
  clients lookup services through a directory in order to find out how and where to invoke them

Service Directory Pattern



Clients use the Directory Service to lookup published service descriptions that will enable them to perform the actual service invocation

Service Directory Hints

- Clients cannot directly invoke services, they first need to lookup their endpoint for every invocation.
- Directories are a service in its own right (the difference is that its endpoint should be known in advance by all clients).
- The directory data model heavily depends on the context where the directory is applied and to the common assumptions shared by clients and services:
  - Simplest model: map <Symbolic Name> to <IP:Port>
  - Directories can store complete service descriptions or only store references to service descriptions
  - Depending on the metadata stored in the directory, clients can perform sophisticated queries over the syntax and semantics of the published services
- Directories are a critical point for the performance fault-tolerance of the system. Clients usually cache previous results and repeat the lookup only on failure of the invocations.
- The amount of information stored in a directory may grow quite large over time and needs to be validated and kept up to date. Published services should be approved in order for clients to trust the information provided by the directory.

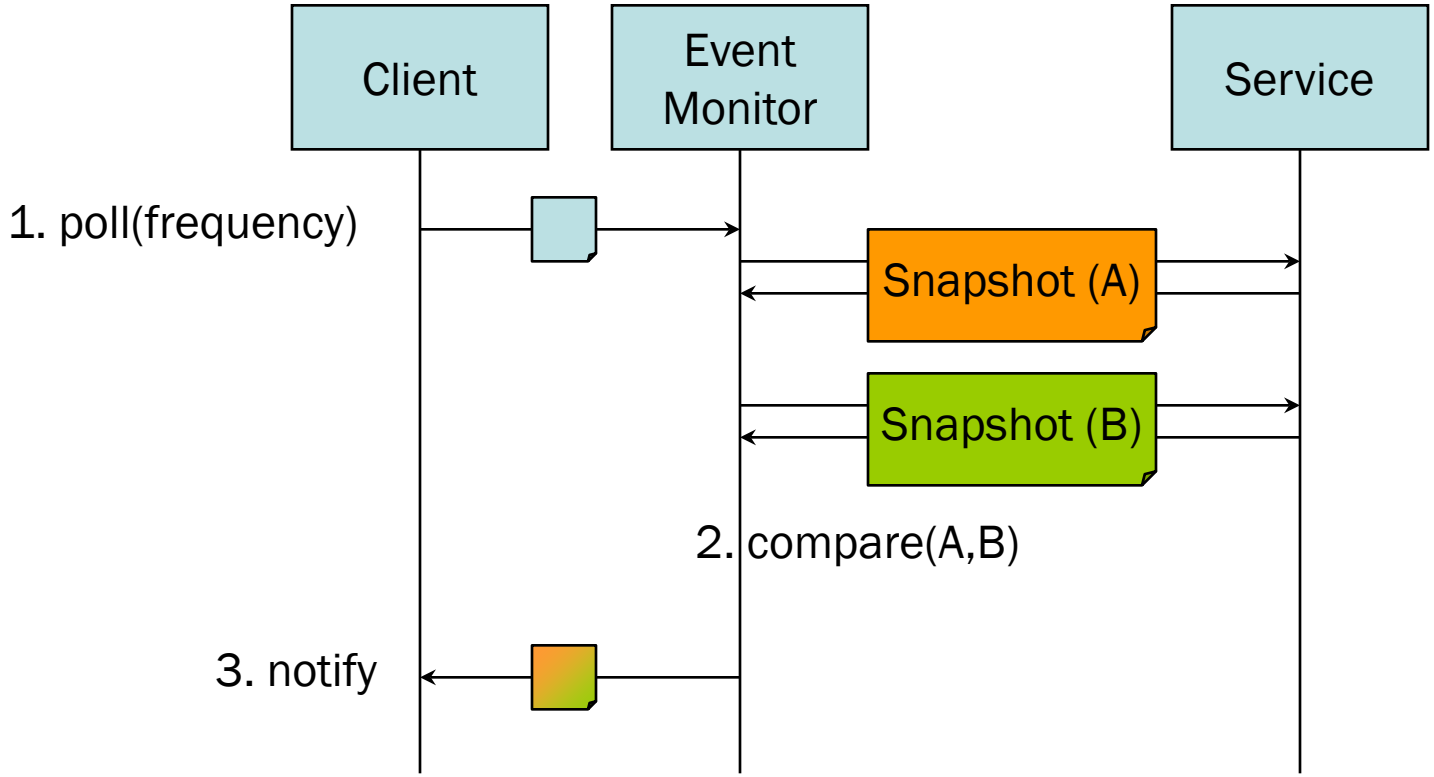
Notification

Event Monitor
Observer
Publish/Subscribe
Messaging Bridge

Event Monitor

- Goal:
  inform clients about events happening at the service
- Constraint:
  service does not support a publish/subscribe mechanism
- Solution:
  poll and compare snapshots
- Pattern:
  clients use an event monitor that periodically polls the service, compares its state with the previous one and notifies the clients about changes

Event Monitor Pattern



A client instructs the Event Monitor to periodically poll the state of a remote service. The Event Monitor only notifies the client if a change has occurred between different snapshots.

Event Monitor: Hints

- Necessary only if a service (or a component) does not provide a publish/subscribe interface
- To minimize the amount of information to be transferred, snapshots should be tailored to what events the client is interested in detecting. This may be a problem if the service interface cannot be changed.
- One event monitor should be shared among multiple clients by using the observer pattern
- Clients should be able to control polling frequency to reduce the load on the monitored service
- Warning: this solution does not guarantee that all changes will be detected, but only those which occur at most as often as the sampling rate of the event monitor

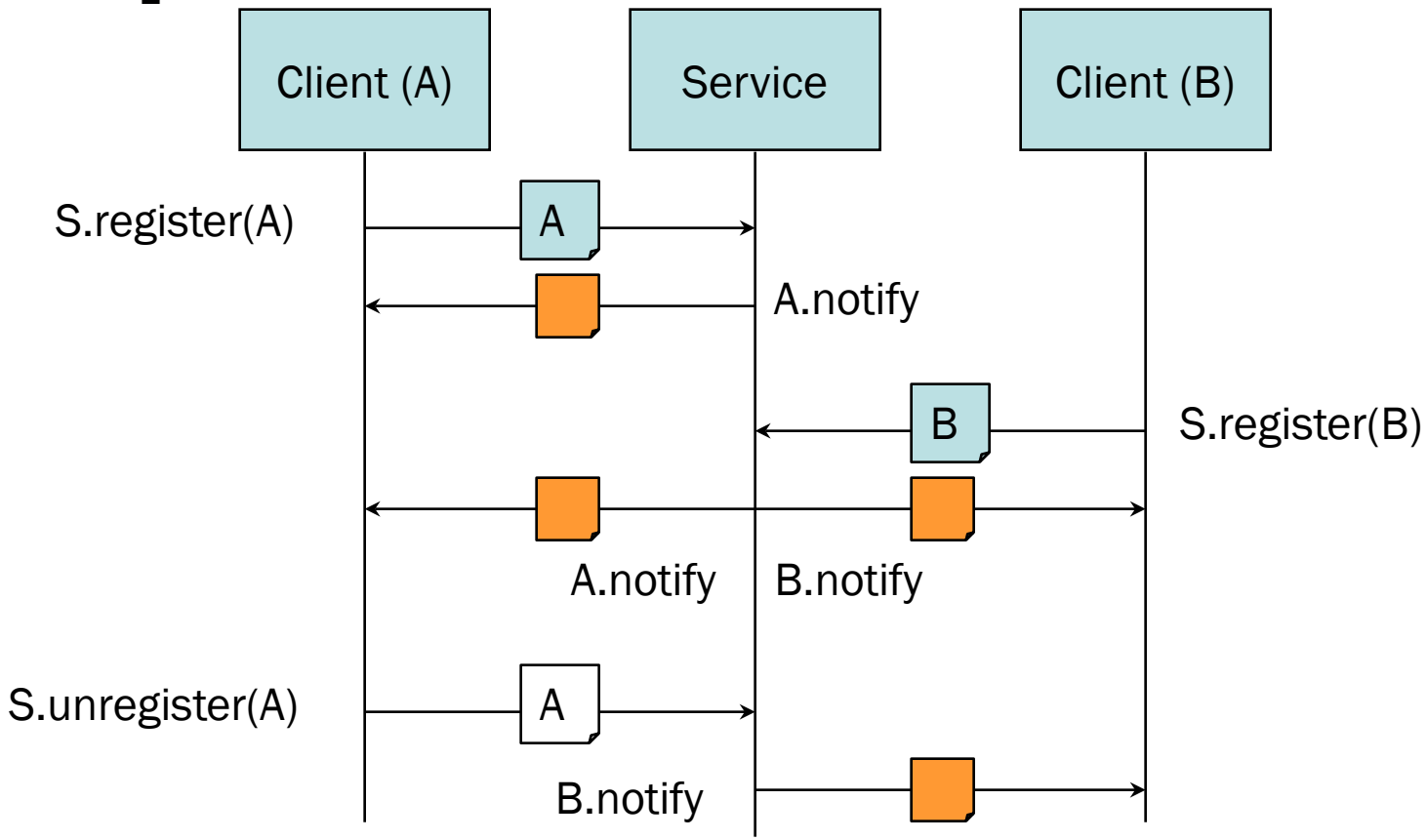
Observer

- Goal:
  ensure a set of clients gets informed about all state changes of a service as soon as they occur

- Solution:
  detect changes and generate events at the service

- Pattern:
  clients register themselves with the service, which will inform them whenever a change occurs

Observer Pattern



Clients register a callback/notification endpoint with the service.
This is used by the service to notify all clients about state changes.
Clients should also unregister themselves when no longer interested.

Observer Hints

- Assuming that a service can implement it, this pattern improves on the Event Monitor pattern:
  - State changes are not downsampled by clients
  - State changes are propagated as soon as they occur
  - If no change occurs, useless polling by clients is spared
- Clients could share the same endpoint with several observed services. Notifications should identify the service from which the event originates.
- To avoid unnecessary polling by the client, notification messages should also include information about the new state and not just that a state change has occurred.
- Warning: the set of registered clients becomes part of the state of the service and may have to be made persistent to survive service failures.
- A client which stops listening without unregistering should not affect the other registered clients

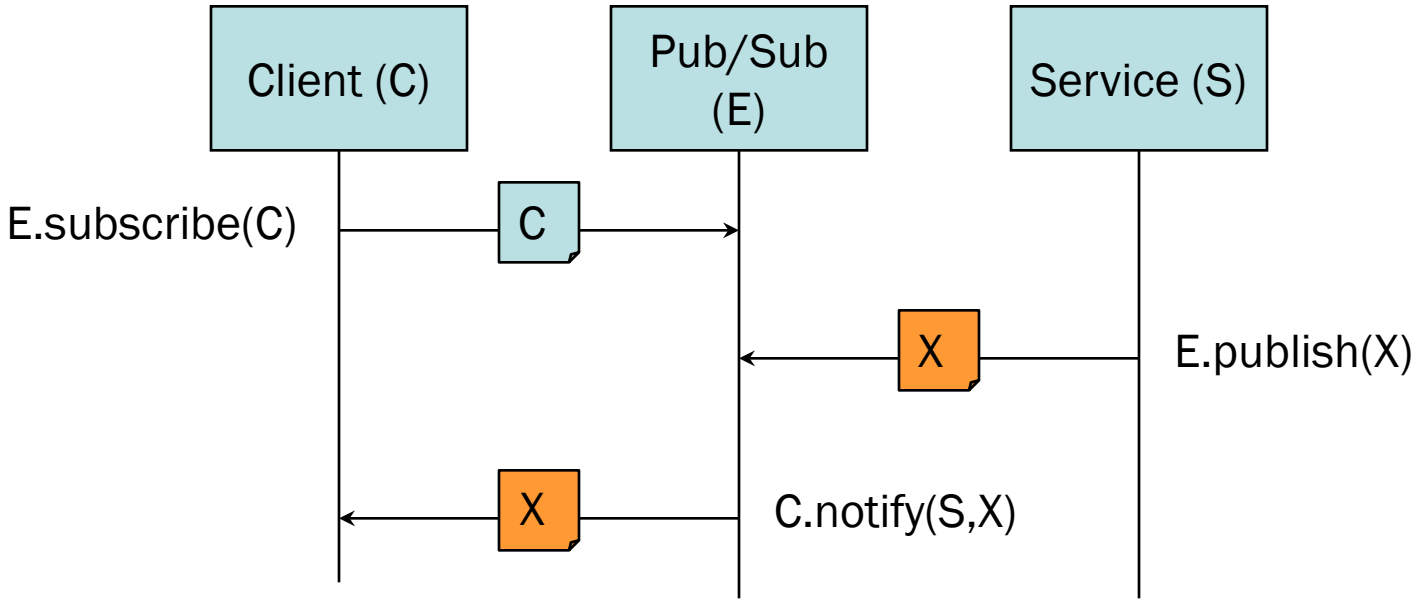
Publish/Subscribe

- Goal:
  decouple clients from services generating events

- Solution:
  factor out event propagation and subscription management into a separate service

- Pattern:
  clients register with the event service by subscribing to certain event types, which are published to the event service by a set of one or more services.

Publish/Subscribe Pattern



As opposed to the Observer, subscriptions are centrally managed for a set of services by the Pub/Sub Service.
As opposed to the Event Monitor, services actively push state changes into the Pub/Sub Service.

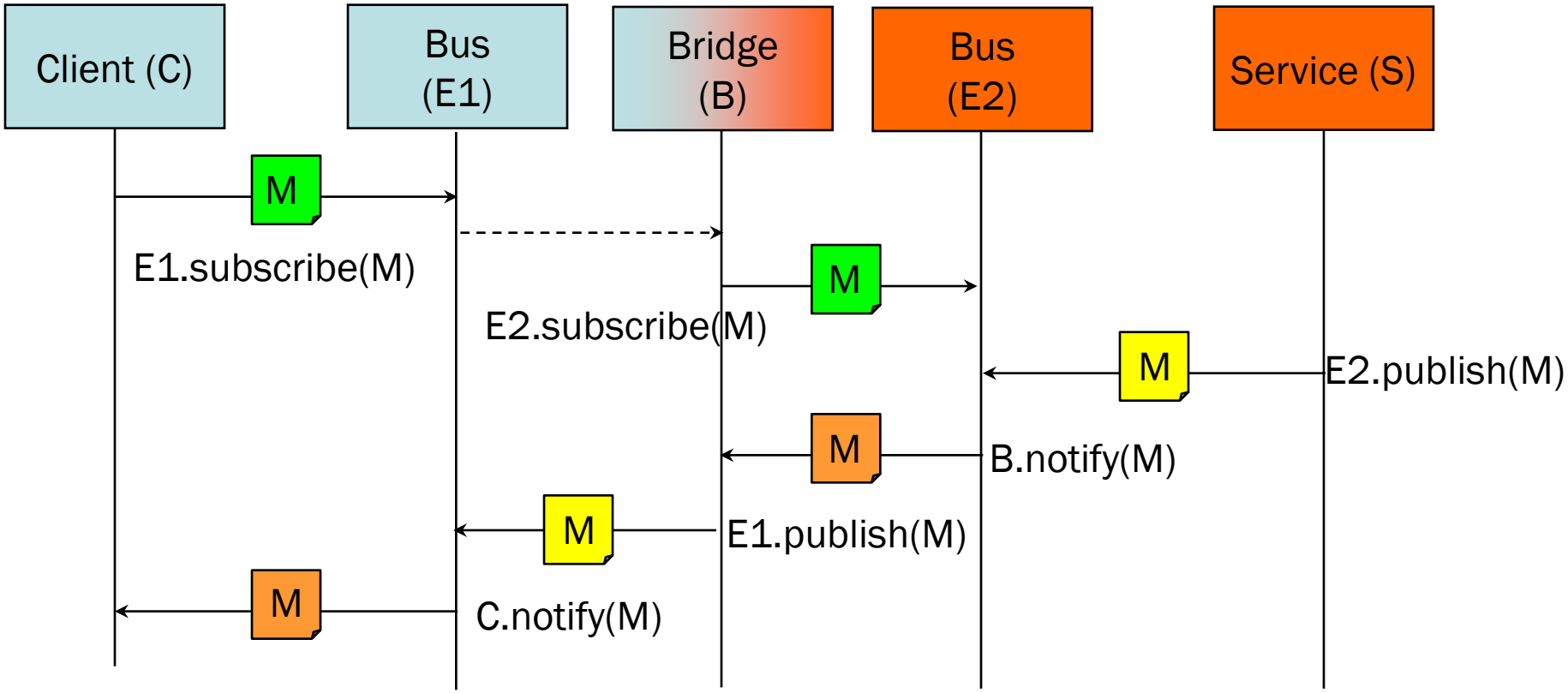
Publish/Subscribe Hints

- The interaction between published and subscriber is similar to the Observer pattern. It is also more decoupled, as messages are routed based on their type.
- Warning: event types become part of the interface of a service, and it may be difficult to determine the impact of changes in event type definitions
- Like in the Observer pattern, subscriptions should be made persistent, as it is difficult for clients to realize that the pub/sub service has failed.
- Unlike with the Observer pattern, all events in the system go through a centralized component, which can become a performance bottleneck. This can be alleviated by partitioning the pub/sub service by event type.

Messaging Bridge

- Goal:
  connect multiple messaging systems
- Solution:
  link multiple messaging systems to make messages exchanged on one also available on the others
- Pattern:
  introduce a bridge between the messaging systems, the bridge forwards (and converts) all messages between the buses

Messaging Bridge Pattern



The bridge (B) replays all subscriptions made by clients one bus (E1) onto the other (E2) so that it can inform the client (C) of one bus of messages published on the other

Messaging Bridge Hints

- Enable the integration of existing multiple messaging systems so that applications do not have to use multiple messaging systems to communicate
- Often it is not possible to use a single messaging system (or bus) to integrate all applications:
  - Incompatible messaging middleware (JMS vs. .NET MQ)
  - External bus(es) vs. Internal bus
  - One bus does not scale to handle all messages
- Although messaging systems may implement the same standard API, they are rarely interoperable so they cannot be directly connected.
- Messaging bridges are available (or can be implemented) so that messages can be exchanged between different buses. They act as a message consumer and producer at the same time and may have to perform message format translation.

Composition

Scatter/Gather

Master/Slave

Synchronous

Asynchronous

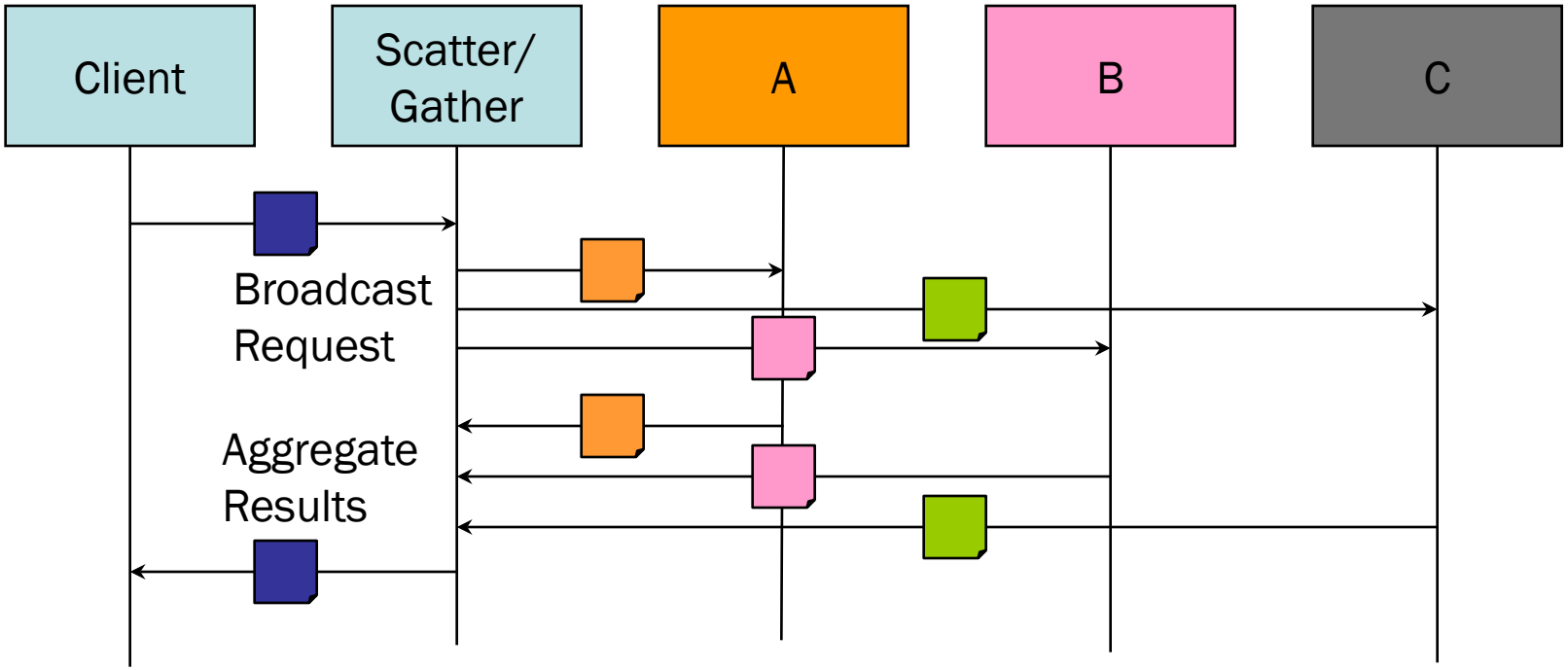
Scatter/Gather

- Goal:
  send the same message to multiple recipients which will (or may) reply to it

- Solution:
  combine the notification of the request with aggregation of replies

- Pattern:
  broadcast a message to all recipients, wait for all (or some) answers and aggregate them into a single message.

Scatter/Gather Pattern



Broadcasting can be implemented using subscriptions (loosely coupled) or a distribution list provided by the client (which knows A, B, C)
Results will be collected and aggregated before they are returned to the original client as a single message

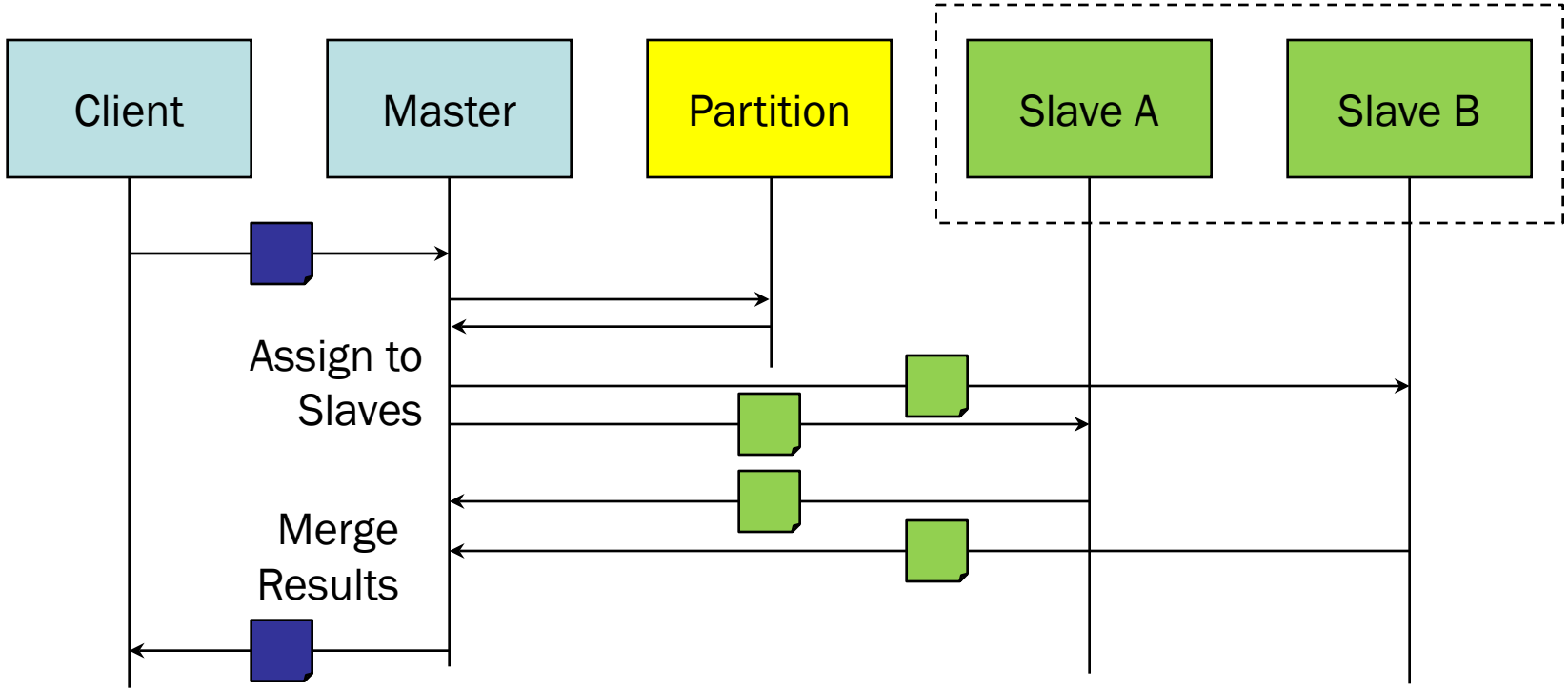
Scatter/Gather Hints

- This is a simple composition pattern, where we are interested in aggregating replies of the components processing the same message.
- Alternatives for controlling the components:
  - The recipients can be **discovered** (using subscriptions) or be **known a priori** (distribution list attached to request)
- Warning: the response-time of each component may vary and the response-time of the scatter/gather is the slowest of all component responses
- Different synchronization strategies:
  - Wait for all messages
  - Wait for some messages (within a certain time window, or using an N-out-of-M synchronization)
  - Return fastest reply (*discriminator*, maybe only under certain conditions)
- Example:
  - Contact N airlines simultaneously for price quotes
  - Buy ticket from either airline if price<=200 CHF
  - Buy the cheapest ticket if price >200 CHF
  - Make the decision within 2 minutes

Master/Slave

- Goal:
  speed up the execution of long running computations
- Solution:
  split a large job into smaller independent partitions which can be processed in parallel
- Pattern:
  the master divides the work among a pool of slaves and gathers the results once they arrive

Master/Slave Pattern



Assignment to slaves can be implemented using push or pull.
The set of slaves is not known in advance and may change over time.
Results will be collected and aggregated before they are returned to the original client

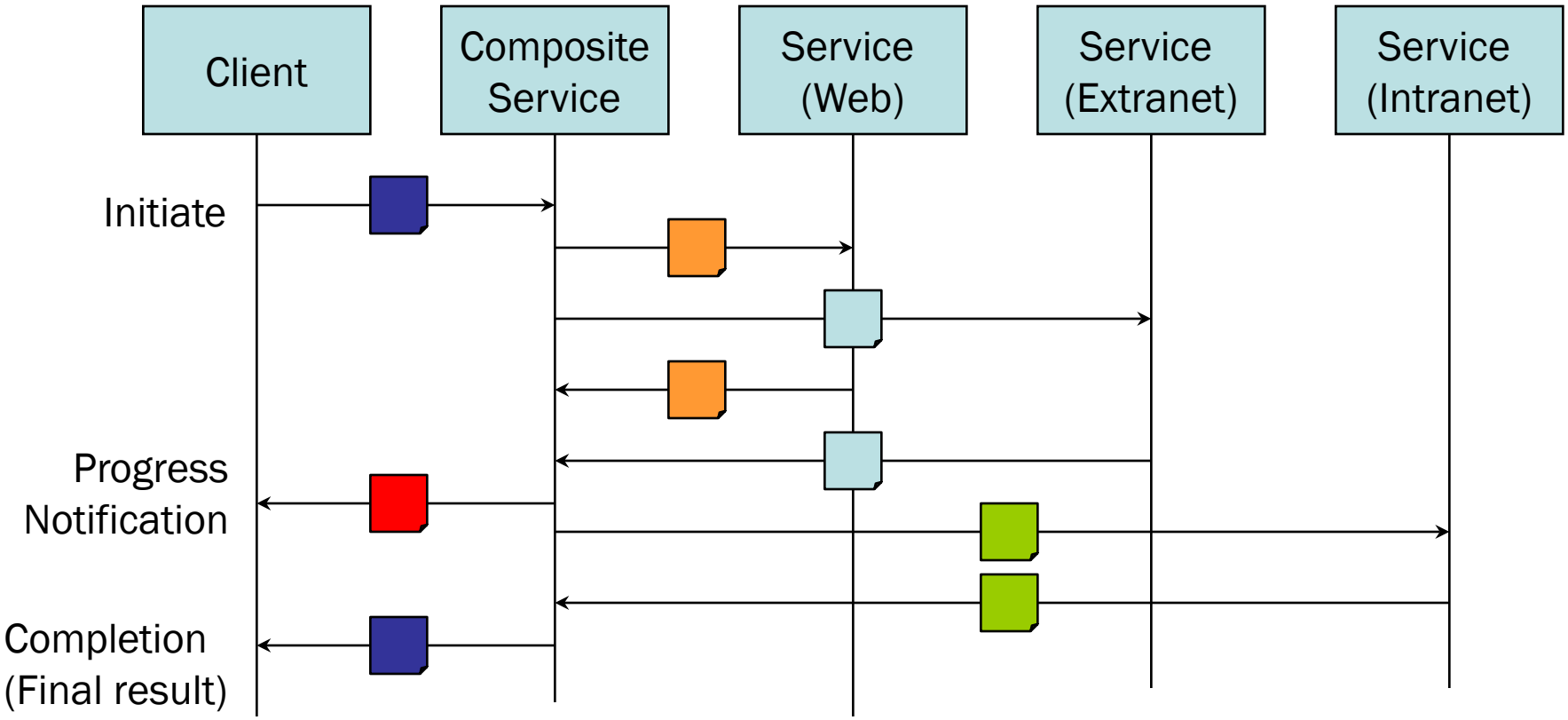
Master/Slave Hints

- This composition pattern is a specialized version of the scatter/gather pattern, also known as *divide and conquer*
- Clients should not know that the master delegates its task to a set of slaves
- Master Properties:
  - Different partitioning strategies may be used
  - Fault Tolerance: if a slave fails, resend its partition to another one
  - Computational Accuracy: send the same partition to multiple slaves and compare their results to detect inaccuracies (this works only if slaves are deterministic)
  - Master is application independent
- Slave Properties:
  - Each slave runs its own thread of control
  - Slaves may be distributed across the network
  - Slaves may join and leave the system at any time (may even fail)
  - Slaves do not usually exchange any information among themselves
  - Slaves should be independent from the algorithm used to partition the work
- Example Applications:
  - Matrix Multiplication (compute each row independently)
  - Movie Rendering (compute each picture frame independently)
  - TSP Combinatorial Optimization (local vs. global search)

Composition

- Goal:
  Improve reuse of existing applications
- Solution:
  Build systems out of the composition of existing ones
- Pattern:
  by including compositions as an explicit part of the architecture, it becomes possible to reuse existing services in order to aggregate basic services into value-added ones
- Variants: Synchronous, Asynchronous
- Synonyms: Orchestration

Composition Pattern



Composite services provide value-added services to clients by aggregating a set of services and orchestrating them according to a well-defined and explicit business process model

Composition Hints

- Composition is recursive: a composite service is a service that can be composed.
- Services involved in a composition do not necessarily have to know that they are being orchestrated as part of another service and may be provided by different organizations
- Services can be reused across different compositions and compositions can be reused by binding them to different services
- Composite services may be implemented using a variety of tools and techniques:
  - Ordinary programming languages (Java, C#, …)
  - Scripting languages (Python, PERL, …)
  - Workflow modeling languages (JOpera, BPEL…)
- Compositions may be long-running processes and involve asynchronous interactions (both on the client side, and with the services)

Università
della
Svizzera
italiana

**Facoltà
di scienze
informatiche**

Testing

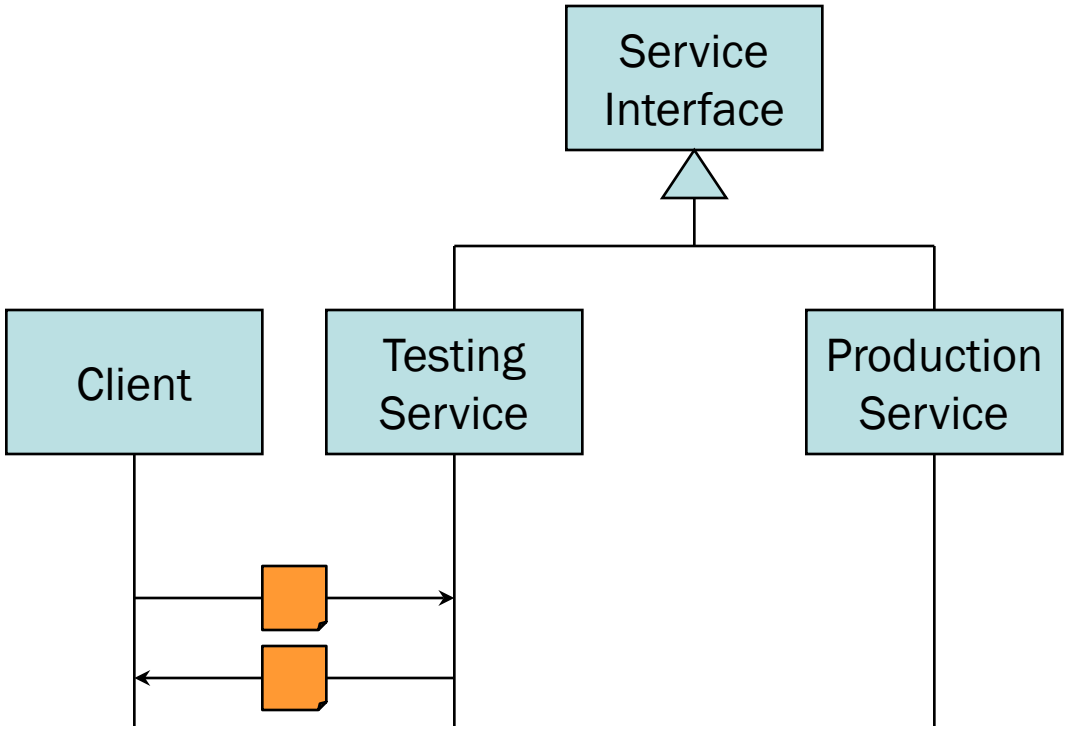
Faux Implementation
Wire Tap

Faux Implementation

- Goal:
  test a client separately from the services it depends on
- Solution:
  bind the client to a local replacement
- Pattern:
  the interfaces of the production services that are needed by the client are implemented by mockup services. These are used to test & debug the client that does not have to call the remote service

- Synonyms: Interface Pattern, Mockup

Faux Implementation Pattern



For testing the client calls a replacement service which implements the same interface as the production one.

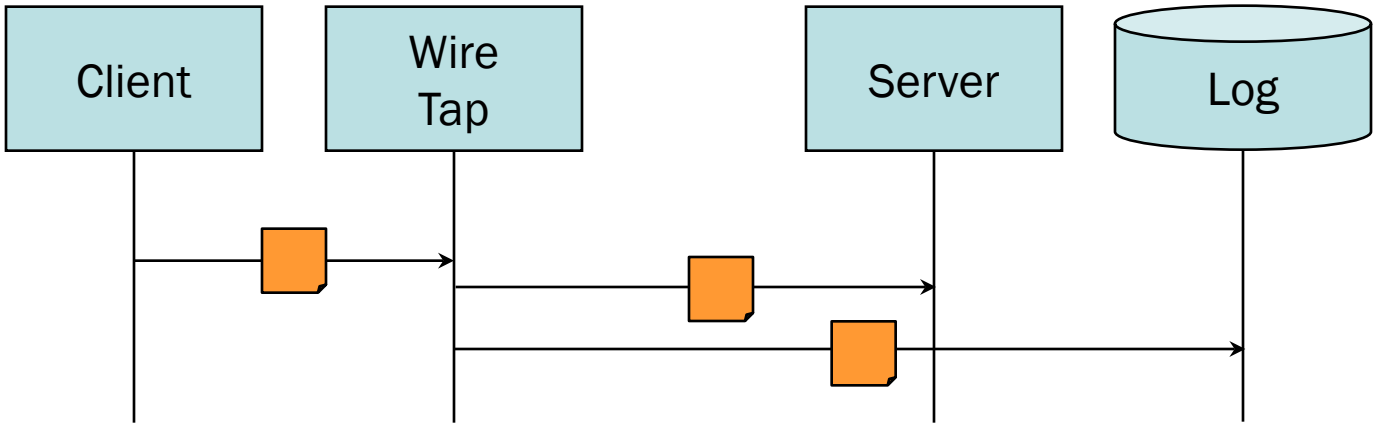
Faux Implementation: Hints

- Necessary when the production Web services are not (yet) available for testing the client
- Warning: the faux implementation should behave like the real one it replaces:
  - Compatible functional interface
  - Similar non-functional properties: e.g., response time
- Applicable when it is good enough to re-implement a subset of the original service interface.
- As an alternative, a copy of the production service could be redeployed for testing

Wire Tap

- Goal:
  inspect messages between client and server
- Solution:
  intercept messages in transit using a “wire tap”
- Design Pattern:
  The wire tap replaces the intended destination of the messages, forwards them to the original destination but also keeps a copy for inspection
- Synonyms: Tee

Wire Tap Pattern



The client may have to be modified to send messages to the wiretap
The server may be transparently replaced by the wiretap
Wire tap acts as a proxy between client and server

Wire Taps: Hints

- Useful for debugging (read the data on the wire), auditing (log all information in transit), performance measurement (measure the time between the request and response message)
- Warning: wire taps introduce additional latency & overhead, especially if the messages are logged synchronously
- Three ways to set it up:
  - Can Modify the Client: point the client to the wire tap proxy instead of the original destination.
  - Can Replace the Server: make the wiretap listen on the original server endpoint and forward messages to new server address.
  - Cannot Change Anything: use a packet sniffer.

Università
della
Svizzera
italiana

**Facoltà
di scienze
informatiche**

Optimization

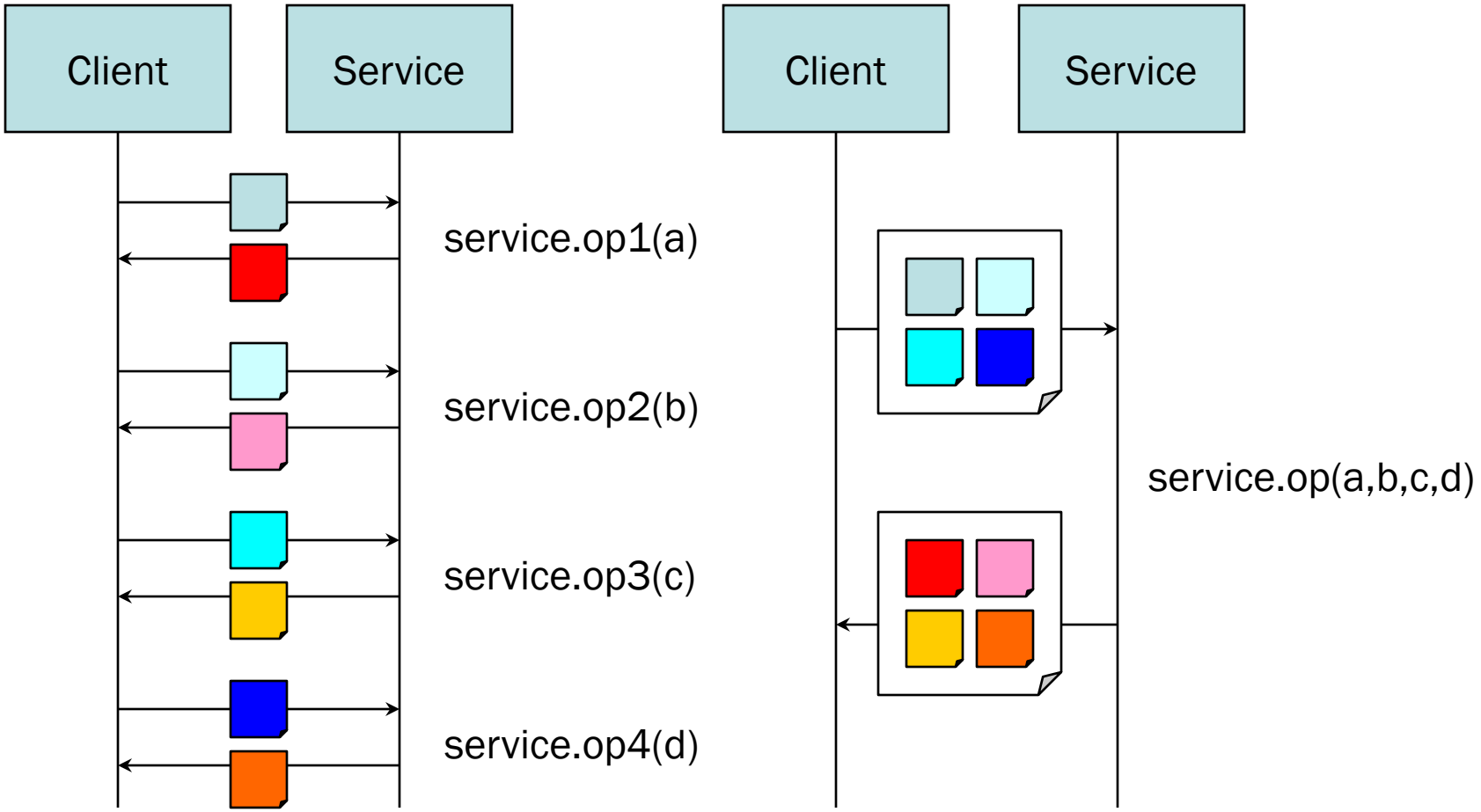
Data Transfer Object

Partial Population

Data Transfer Object

- Goal:
  reduce load on service, minimize response time (Performance Optimization)
- Solution:
  reduce the number of messages that are exchanged
- Pattern:
  operations should be designed with the optimal level of granularity to perform a given function. The required input information and the results should be exchanged with a minimal number of messages (2 in the best case)

Data Transfer Object Pattern



Data Transfer Object: Hints

- Performance is improved if:

$$N * O(Call(1)) > O(Call(N))$$

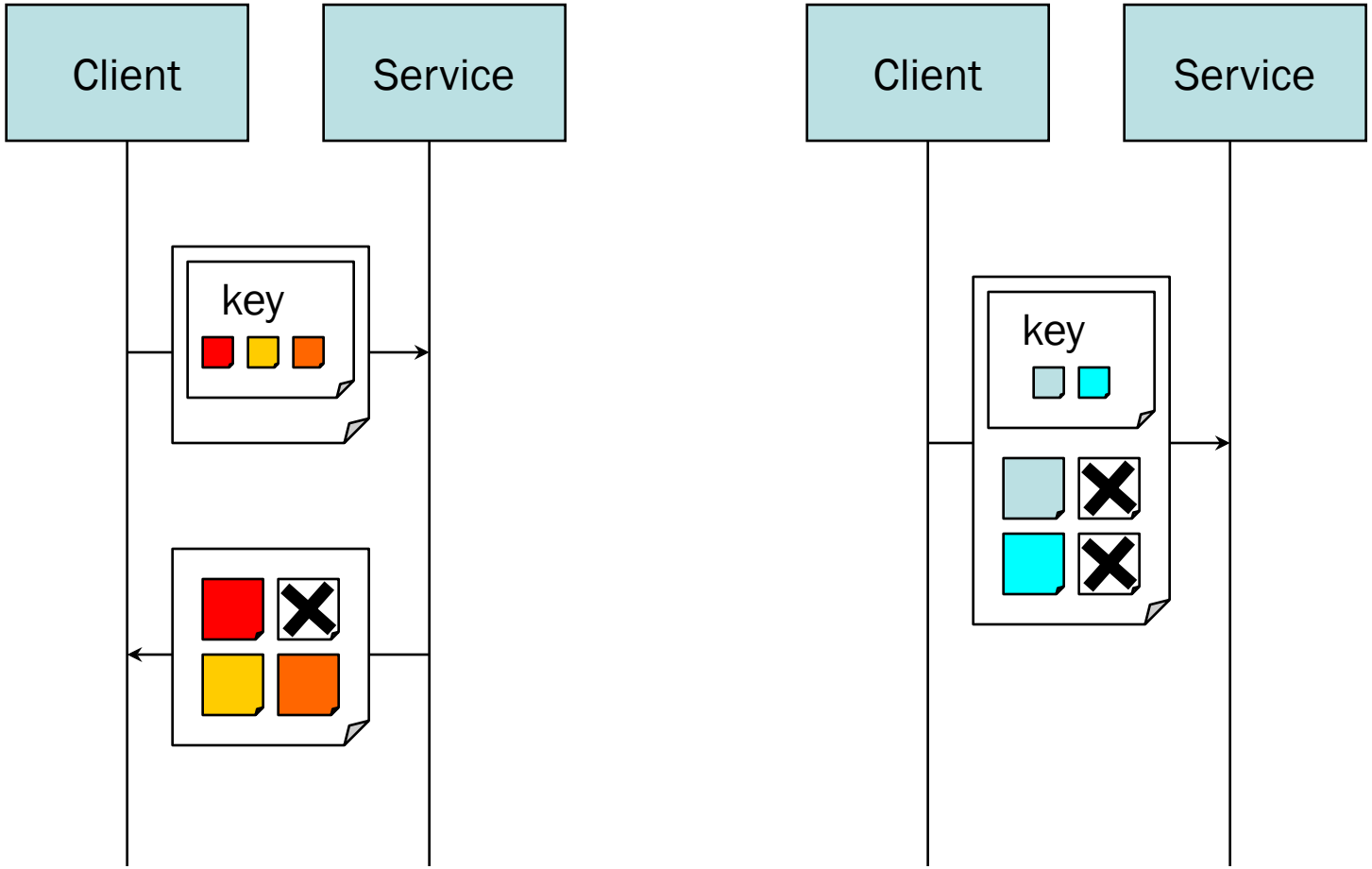
Calling N-times an operation with one object takes longer than calling once an operation passing N-objects

- Applicable when clients retrieve multiple data items from a service
- Warning: a Data Transfer Object should not have behavior associated with it
- Can be used to ensure client gets a consistent snapshot of data on server (if each Call runs in its own transaction)
- Variation: Data Transfer Collection (when multiple data objects are transferred with one message)

Partial Population

- Goal:
  control and reduce message size
- Solution:
  a further optimization of the Data Transfer Object pattern, where the amount of data to be transferred can be controlled by the client
- Pattern:
  extend a Data Transfer Object with a key field – explicit description of the actual data it contains (for sending data to service) and pass the key as input to operation (for choosing what the service should return)

Partial Population Pattern



Filtering Partial Results

Sending Partial Requests

Partial Population: Hints

- Advantage: Avoid defining Data Transfer Objects for every combination of data items that a client may ever need
- The same Data Transfer Object can use “null” values to avoid transferring unwanted fields and reduce message sizes
- Further generalizing this pattern leads to a service providing a flexible “query” (RESTful) interface, where clients use the same operation to submit any possible query over the data provided by the service

ResponseDocument = Query(RequestDocument)

Response = Call(Method, Request)

Some Advice

The best architectures are full of patterns

Do not use too many unnecessary patterns (this is an anti-pattern)

More Advice

Most software systems cannot be structured according to a single architectural pattern

Choosing one or more patterns does not give a complete software architecture. Further refinement is needed

From Marek Reformat

References

- Paul Monday, **Web Service Patterns: Java Edition**, APress, 2003
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, **Design Patterns: Elements of Reusable Object-Oriented Software**, Addison-Wesley, 1994
- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, **Patterns-Oriented Software Architecture, Vol. 1: A System of Patterns,** John Wiley, 1996
- Martin Fowler, **Patterns of Enterprise Application Architecture**, Addison-Wesley, 2003
- Gregor Hohpe, Bobby Woolf, **Enterprise Integration Patterns**, Addison Wesley, 2004
- William Brown, Raphael malveau, Hays McCormik III, Thomas Mowbray, **Anti-Patterns, Refactoring Software, Architectures, and Projects in Crisis**, Wiley, 1998
  http://sourcemaking.com/antipatterns