CSC301 Introduction to Software Architecture

Alexei Lapouchnian

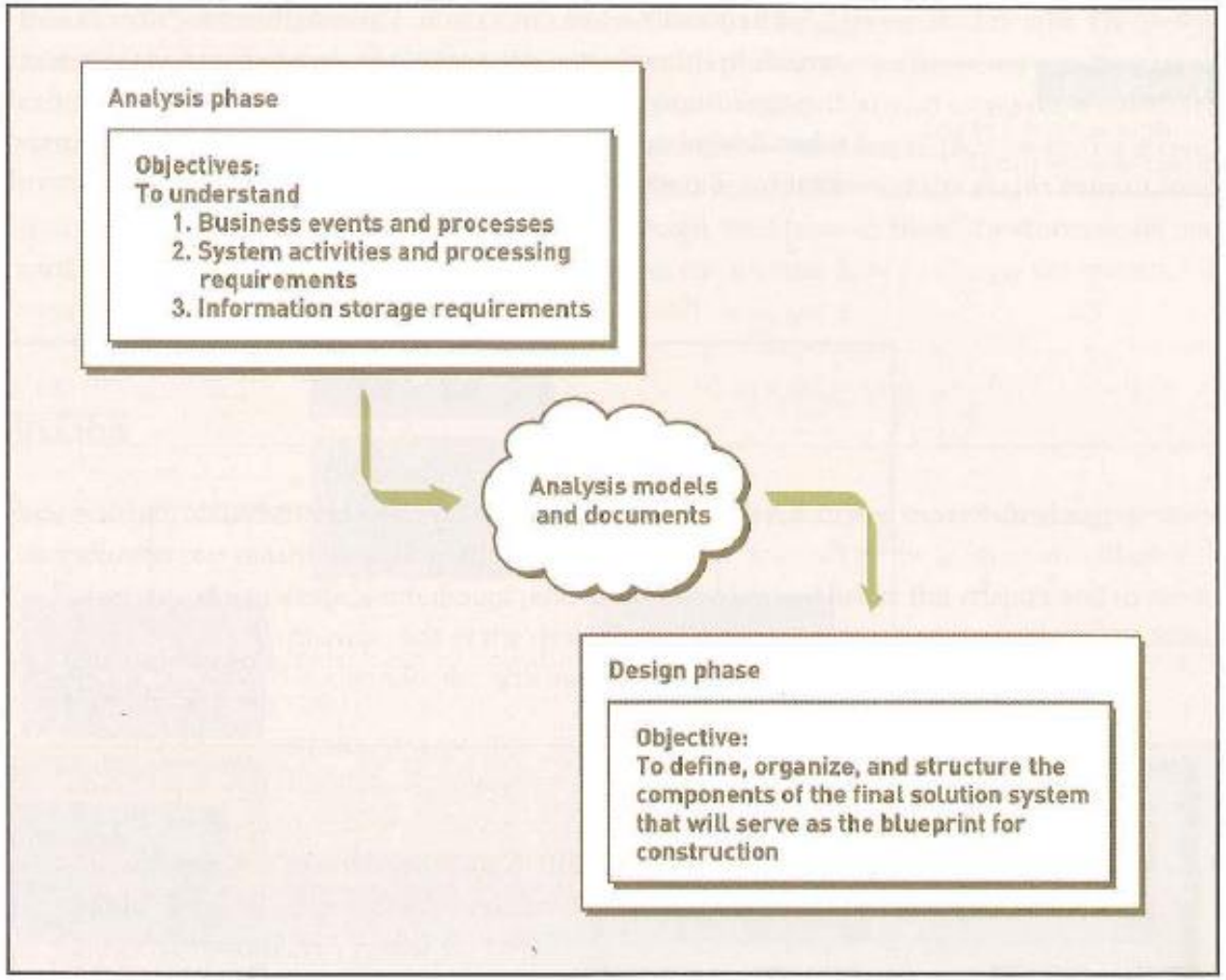
9

Introduction to Software Architectures

Material by Len Bass, Paul Clements, Rick Kazman, Humberto Cervantes, Nick Rozanski, and Eoin Woods

Architectural Design

- Satzinger



References

- Bass, L., Clements, P., & Kazman, R. (2013). *Software architecture in practice, 3rd Ed.* Addison-Wesley. **(BCK)**
  - Available through Safari Books Online (TPL):
    - http://proquestcombo.safaribooksonline.com.ezproxy.torontopubliclibrary.ca/book/software-engineering-and-development/9780132942799
- Cervantes, H., & Kazman, R. (2016). Designing software architectures: a practical approach. Addison-Wesley. **(CK)**
  - Available through Safari Books Online (TPL):
    - http://proquestcombo.safaribooksonline.com.ezproxy.torontopubliclibrary.ca/book/software-engineering-and-development/9780134390857
- Rozanski, B. & Woods, E. (2011). *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives* (2nd edition). Addison-Wesley. **(RW)**
  - Available through Safari Books Online (TPL):
    - http://proquestcombo.safaribooksonline.com.ezproxy.torontopubliclibrary.ca/book/software-engineering-and-development/9780132906135

What is Software Architecture?

- The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both (BCK)
- The architecture of a system is the set of fundamental concepts or properties of the system in its environment, embodied in its elements, relationships, and the principles of its design and evolution (RW)

Importance of Software Architecture

Importance of Software Architecture [1] [CK]

- An architecture will inhibit or enable a system's driving **quality attributes**.
- The decisions made in an architecture allow you to reason about and manage change as the system evolves.
- The analysis of an architecture enables early prediction of a system's qualities.
- A documented architecture enhances communication among stakeholders.
- The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
- An architecture defines a set of constraints on subsequent implementation.
- The architecture influences the structure of an organization, or vice versa.

Importance of Software Architecture [2] [CK]

- An architecture can provide the basis for evolutionary, or even throwaway, prototyping.
- An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule.
- An architecture can be created as a transferable, reusable model that forms the heart of a product line.
- Architecture-based development focuses attention on the assembly of components, rather than simply on their creation.
- By restricting design alternatives, architecture channels the creativity of developers, reducing design and system complexity.
- An architecture can be the foundation for training a new team member.

Importance of Software Architecture [1] [CK]

- An architecture will inhibit or enable a system's driving **quality attributes**.
- The decisions made in an architecture allow you to reason about and manage change as the system evolves.
- The analysis of an architecture enables early prediction of a system's qualities.
- A documented architecture enhances communication among stakeholders.
- The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
- An architecture defines a set of constraints on subsequent implementation.
- The architecture influences the structure of an organization, or vice versa.

Importance of Software Architecture [2] [CK]

- An architecture can provide the basis for evolutionary, or even throwaway, prototyping.
- An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule.
- An architecture can be created as a transferable, reusable model that forms the heart of a product line.
- Architecture-based development focuses attention on the assembly of components, rather than simply on their creation.
- By restricting design alternatives, architecture channels the creativity of developers, reducing design and system complexity.
- An architecture can be the foundation for training a new team member.

Inhibiting or Enabling a System's Quality Attributes

- Whether a system will be able to exhibit its desired (or required) quality attributes is substantially determined by its architecture.
- This is the most important message of this course!
  - Performance: You must manage the time-based behavior of elements, their use of shared resources, and the frequency and volume of inter-element communication.
  - Modifiability: Assign responsibilities to elements so that the majority of changes to the system will affect a small number of those elements.
  - Security: Manage and protect inter-element communication and control which elements are allowed to access which information; you may also need to introduce specialized elements (such as an authorization mechanism).
  - Scalability: Localize the use of resources to facilitate introduction of higher-capacity replacements, and you must avoid hardcoding in resource assumptions or limits.
  - Incremental subset delivery: Manage inter-component usage.
  - Reusability: Restrict inter-element coupling, so that when you extract an element, it does not come out with too many attachments to its current environment.

Reasoning About and Managing Change

- About 80 percent of a typical software system's total cost occurs after initial deployment
  - accommodate new features
  - adapt to new environments,
  - fix bugs, and so forth.
- Every architecture partitions possible changes into three categories
  - A *local* change can be accomplished by modifying a single element.
  - A *nonlocal* change requires multiple element modifications but leaves the underlying architectural approach intact.
  - An *architectural* change affects the fundamental ways in which the elements interact with each other and will probably require changes all over the system.
- **Obviously, local changes are the most desirable**
- A good architecture is one in which the most common changes are local, and hence easy to make.

Early Prediction of System Qualities

- If we know that certain kinds of architectural decisions lead to certain quality attributes in a system, we can make those decisions and rightly expect to be rewarded with the associated quality attributes.
- When we examine an architecture we can look to see if those decisions have been made, and confidently predict that the architecture will exhibit the associated qualities.
- The earlier you can find a problem in your design, the cheaper, easier, and less disruptive it will be to fix.

Architecture Ensures Qualities

- Performance
  - Localize critical operations and minimize communications. Use large rather than fine-grained components
- Security
  - Use a layered architecture with critical assets in the inner layers
- Safety
  - Localize safety-critical features in a small number of sub-systems
- Availability
  - Include redundant components and mechanisms for fault tolerance
- Maintainability
  - Use fine-grained, replaceable components

[Sommerville, Software Engineering 7e, Ch 11]

Influencing the Organizational Structure

- Architecture prescribes the structure of the system being developed.
- That structure becomes engraved in the structure of the development project (and sometimes the structure of the entire organization).
- The architecture is typically used as the basis for the work-breakdown structure.
- The work-breakdown structure in turn dictates
  - units of planning, scheduling, and budget
  - inter-team communication channels
  - configuration control and file-system organization
  - integration and test plans and procedures;
  - much more
- The maintenance activity will also reflect the software structure, with teams formed to maintain specific structural elements from the architecture.
- If these responsibilities have been formalized in a contractual relationship, changing responsibilities could become expensive or even litigious.

Enabling Evolutionary Prototyping

- Once an architecture has been defined, it can be analyzed and prototyped as a skeletal system.
  - A skeletal system is one in which at least some of the infrastructure— how the elements initialize, communicate, share data, access resources, report errors, log activity, and so forth—is built before much of the system's functionality has been created.
- This approach aids the development process because the system is executable early in the product's life cycle.
- The fidelity of the system increases as stubs are instantiated, or prototype parts are replaced with complete versions of these parts of the software.
- This approach allows potential performance problems to be identified early in the product's life cycle.
- These benefits reduce the potential risk in the project.

Using Independently Developed Components

- Architecture-based development often focuses on components that are likely to have been developed separately, even independently, from each other.
- The architecture defines the elements that can be incorporated into the system.
- Commercial off-the-shelf components, open source software, publicly available apps, and networked services are example of interchangeable software components.
- The payoff can be
  - Decreased time to market
  - Increased reliability (widely used software should have its bugs ironed out already)
  - Lower cost (the software supplier can amortize development cost across their customer base)
  - Flexibility (if the component you want to buy is not terribly special purpose, it's likely to be available from several sources, thus increasing your buying leverage)

Basis for Training

- The architecture documentation can serve as the first introduction to the system for new project members.
- Module views are excellent for showing someone the structure of a project
  - Who does what, which teams are assigned to which parts of the system, and so forth.
- Component-and-connector views are excellent for explaining how the system is expected to work and accomplish its job.

Design Concepts

- Building blocks and starting points for creating architectures
  - Reference architectures
  - Design patterns
  - Deployment patterns
  - Tactics
  - Existing components

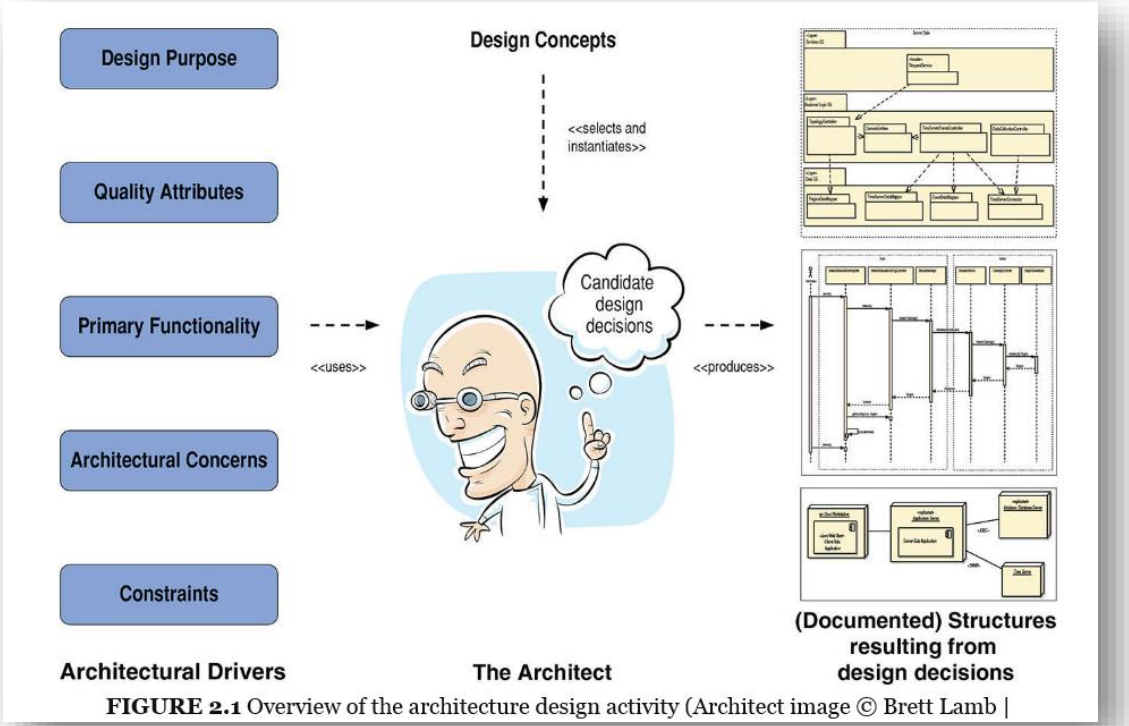


FIGURE 2.1 Overview of the architecture design activity (Architect image © Brett Lamb |

[CK Ch. 2]

Design Concepts

- A Reference Architecture for web applications
- cf:
  - Design patterns
  - Deployment patterns
  - Tactics
  - Existing components

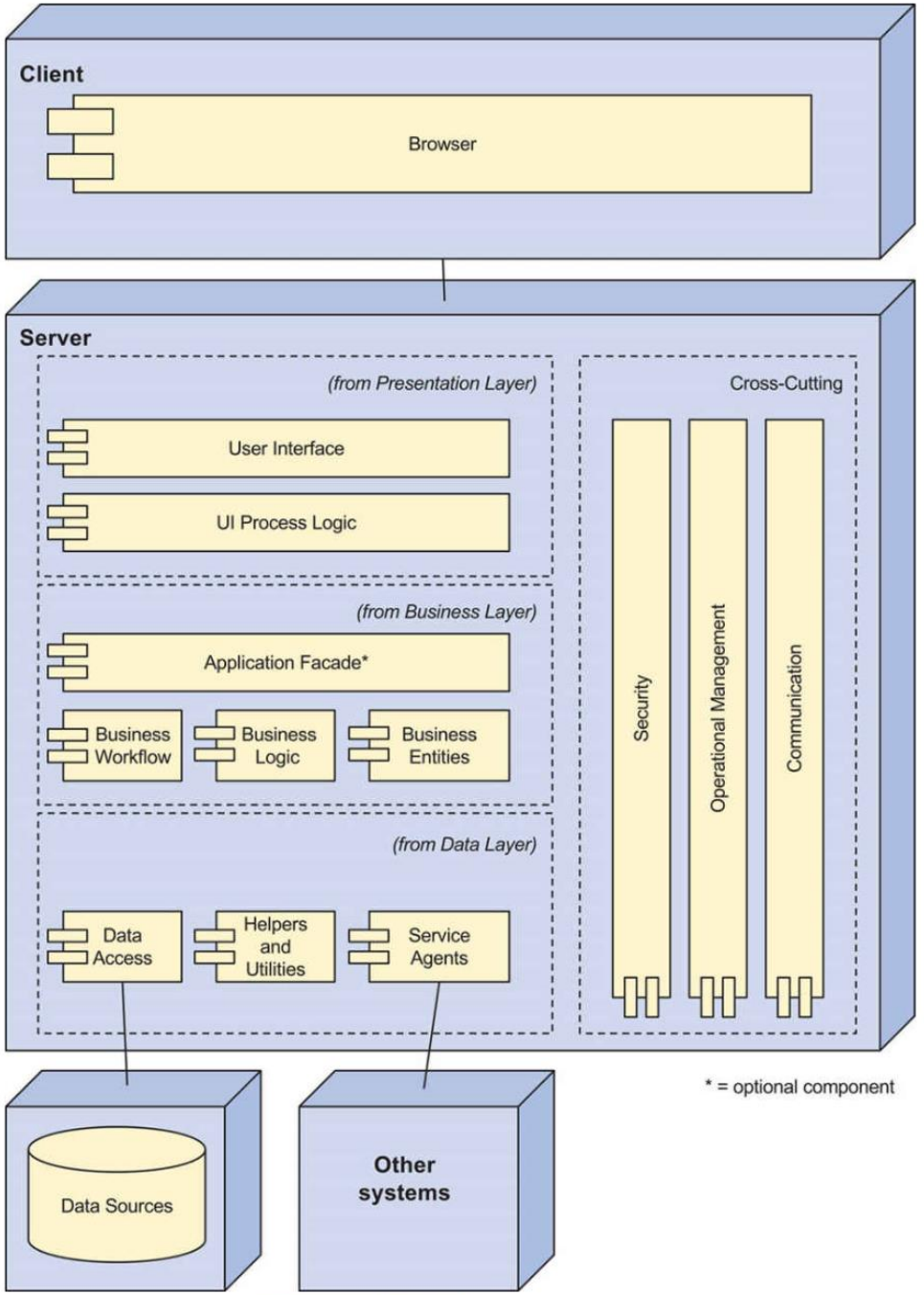


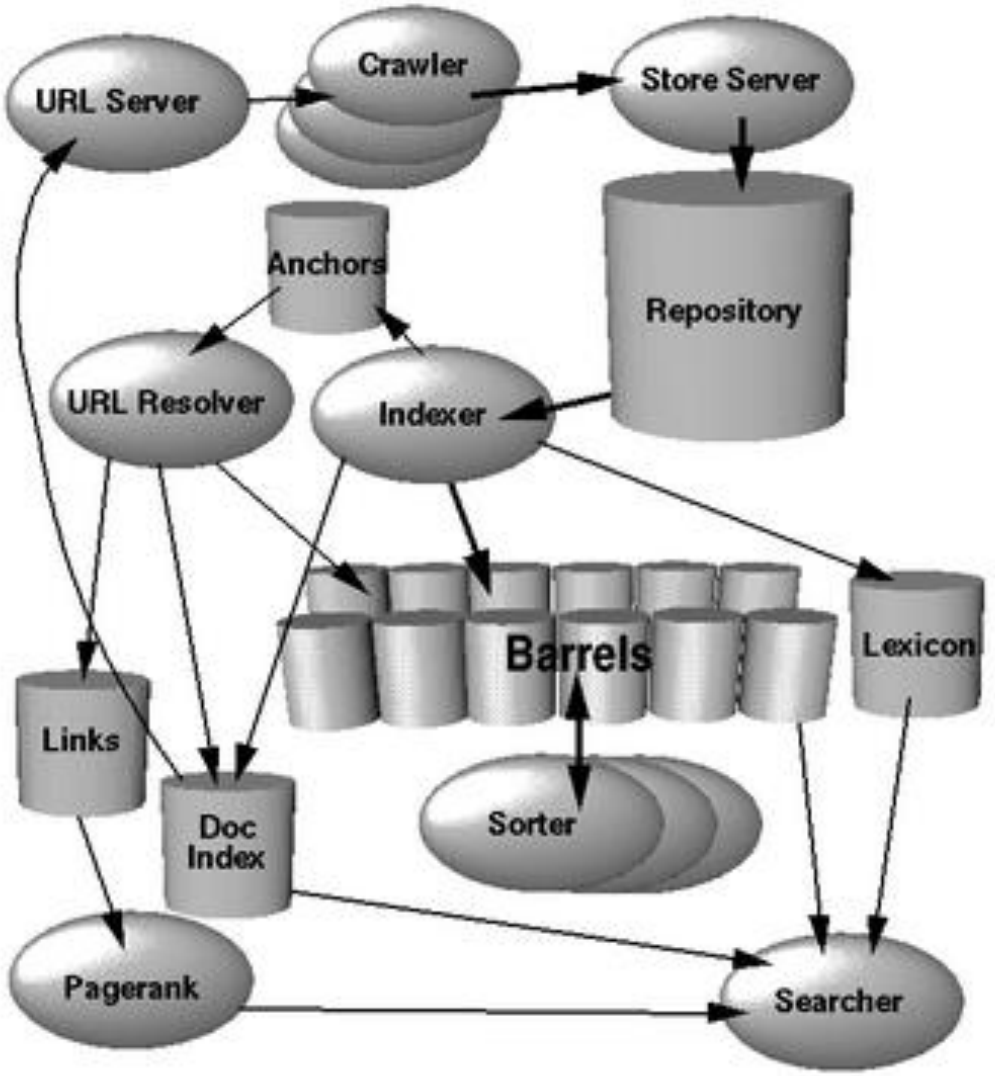
FIGURE 2.3 Example reference architecture for the development of web applications from the *Microsoft Application Architecture Guide* (Key: UML)

- Key concepts from CK ch. 2

Example: A search engine

- **Context**: It's the nineties. Browsing still often starts with directories. But….
- **Vision**: provide a service that allows people to *search* for pages without requiring them to know where to search
- **Architecture**: What components will the system have? Functional decomposition:
  - Crawl pages from the web
  - Index their content and organize it so people can find what they need
  - Provide a search function: result is a set of pages linked to their destination.

A Search Engine…



http://infolab.stanford.edu/~backrub/google.html

4 System Anatomy

First, we will provide a high level discussion of the architecture. Then, there is some in-depth descriptions of important data structures. Finally, the major applications: crawling, indexing, and searching will be examined in depth.

4.1 Google Architecture Overview

In this section, we will give a high level overview of how the whole system works as pictured in Figure 1. Further sections will discuss the applications and data structures not mentioned in this section. Most of Google is implemented in C or C++ for efficiency and can run in either Solaris or Linux.

In Google, the web crawling (downloading of web pages) is done by several distributed crawlers. There is a URLserver that sends lists of URLs to be fetched to the crawlers. The web pages that are fetched are then sent to the storeserver. The storeserver then compresses and stores the web pages into a repository. Every web page has an associated ID number called a docID which is assigned whenever a new URL is parsed out of a web page. The indexing function is performed by the indexer and the sorter. The indexer performs a number of functions. It reads the repository, uncompresses the documents, and parses them. Each document is converted into a set of word occurrences called hits. The hits record the word, position in document, an approximation of font size, and capitalization. The indexer distributes these hits into a set of "barrels", creating a partially sorted forward index. The indexer performs another important function. It parses out all the links in every web page and stores important information about them in an anchors file. This file contains enough information to determine where each link points from and to, and the text of the link.

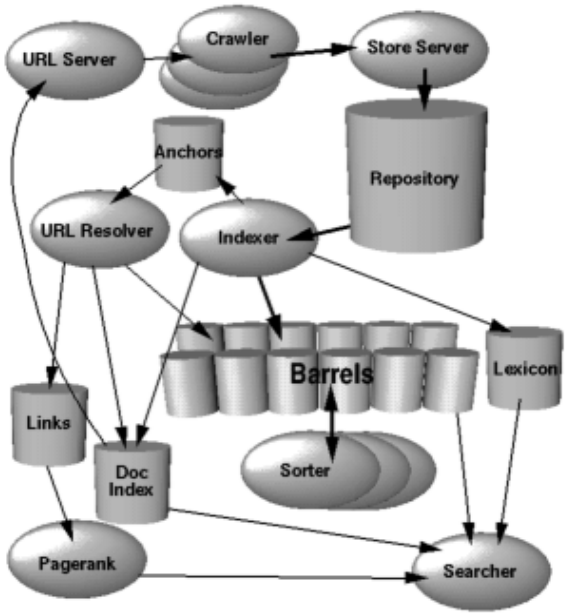


Figure 1. High Level Google Architecture

The URLresolver reads the anchors file and converts relative URLs into absolute URLs and in turn into docIDs. It puts the anchor text into the forward index, associated with the docID that the anchor points to. It also generates a database of links which are pairs of docIDs. The links database is used to compute PageRanks for all the documents.

The sorter takes the barrels, which are sorted by docID (this is a simplification, see Section 4.2.5), and resorts them by wordID to generate the inverted index. This is done in place so that little temporary space is needed for this operation. The sorter also produces a list of wordIDs and offsets into the inverted index. A program called DumpLexicon takes this list together with the lexicon produced by the indexer and generates a new lexicon to be used by the searcher. The searcher is run by a web server and uses the lexicon built by DumpLexicon together with the inverted index and the PageRanks to answer queries.

4.2 Major Data Structures

Structures in Software Architecture

Architecture Is a Set of Software Structures

- A structure is a set of elements held together by a relation.
- Software systems are composed of many structures, and *no single structure holds claim to being the architecture*.
- There are three important categories of architectural structures.
  1. Module
  2. Component and Connector
  3. Allocation

Module Structures (Static)

- Some structures partition systems into implementation units, which we call **modules**.
- Modules are assigned specific computational responsibilities, and are the basis of work assignments for programming teams.
- In large projects, these elements (modules) are subdivided for assignment to sub-teams.

Component-and-connector Structures (Dynamic)

- Other structures focus on the way the elements interact with each other at runtime to carry out the system's functions.
- We call runtime structures **component-and-connector (C&C) structures**.
- In our use, a component is always a runtime entity.
  - Suppose the system is to be built as a set of services.
  - The services, the infrastructure they interact with, and the synchronization and interaction relations among them form another kind of structure often used to describe a system.
  - These services are made up of (compiled from) the programs in the various implementation units – modules.
    - Link to static structures!

Allocation Structures

- **Allocation structures** describe the mapping from software structures to the system's environments
  - Organizational
  - Developmental
  - Installation
  - Execution
- Examples:
  - Modules are 1) assigned to teams to develop, and 2) assigned to places in a file structure for implementation, integration, and testing.
  - Components are deployed onto hardware for execution.

Static Structures [1] [RW]

- The **static structures** of a system define its internal **design-time** elements and their arrangement.
  - Internal software elements: programs, object-oriented classes or packages, database stored procedures, services, or any other self-contained code unit
  - Internal data elements: classes, relational database entities/tables, and data files
  - Internal hardware elements: computers or their constituent parts such as disk or CPU
  - Internal networking elements: cables, routers, or hubs

Static Structures [2] [RW]

- The static arrangement of these elements defines (depending on the context) the associations, relationships, or connectivity between these elements
  - For software modules:
    - A hierarchy of elements (module A is built from modules B and C)
    - Dependencies between elements (module A relies on the services of module B)
  - For data elements (classes, relational entities, etc.): how one data item is linked to another one.
  - For hardware: the required physical interconnections between the various hardware elements of the system.

Static Structure Example [1] [RW]

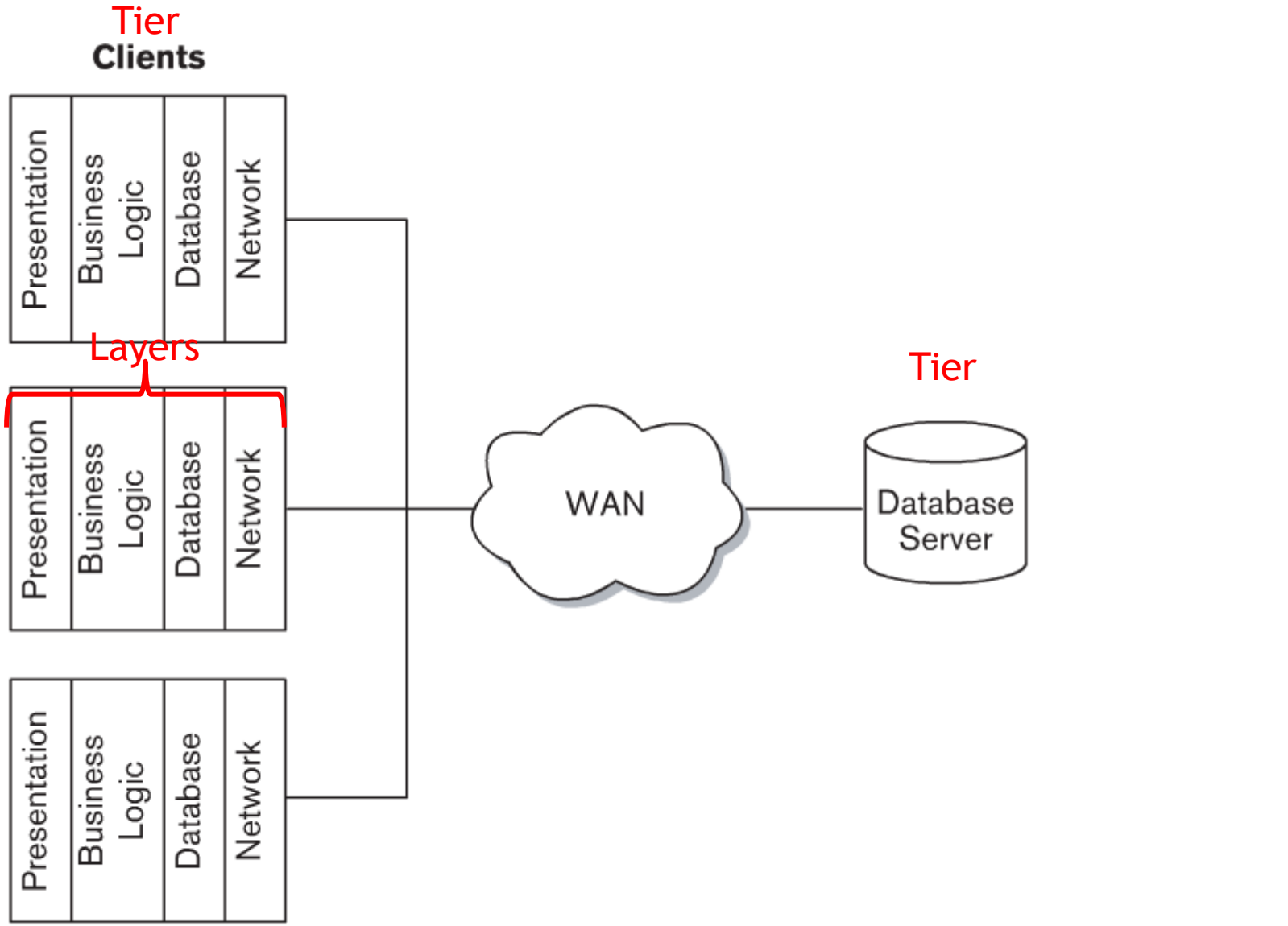


FIGURE 2–2 TWO-TIER CLIENT/SERVER ARCHITECTURE FOR AN AIRLINE BOOKING SYSTEM

Static Structure Example [2] [RW]

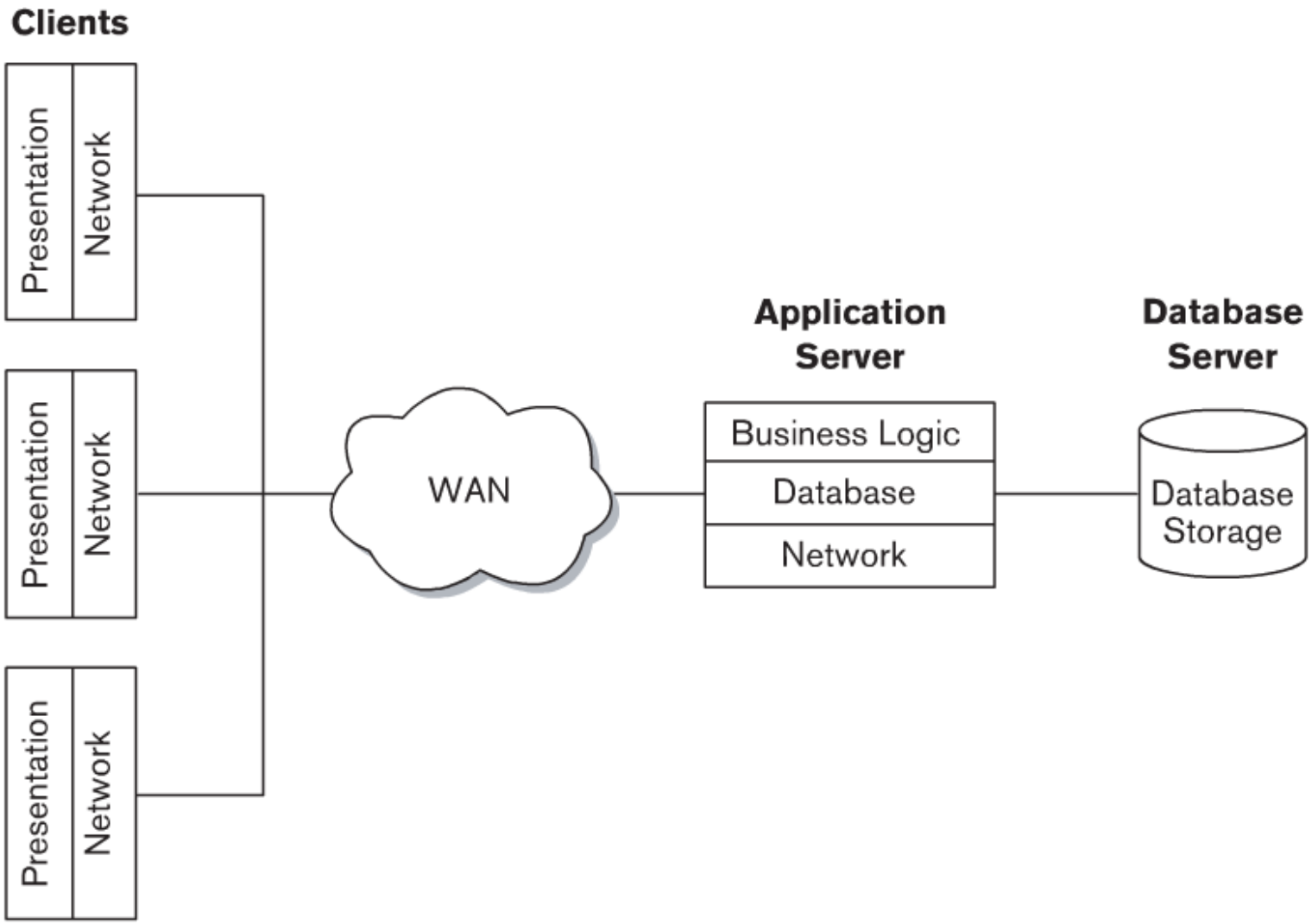
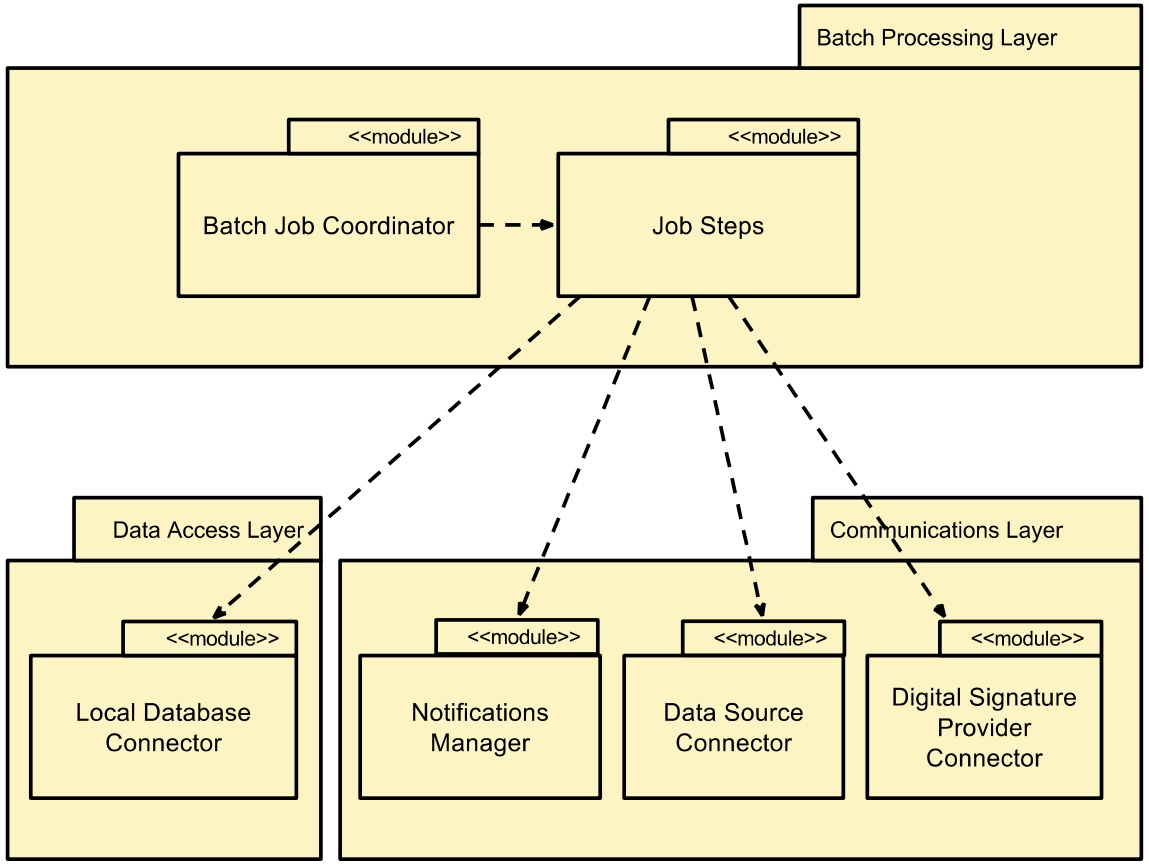


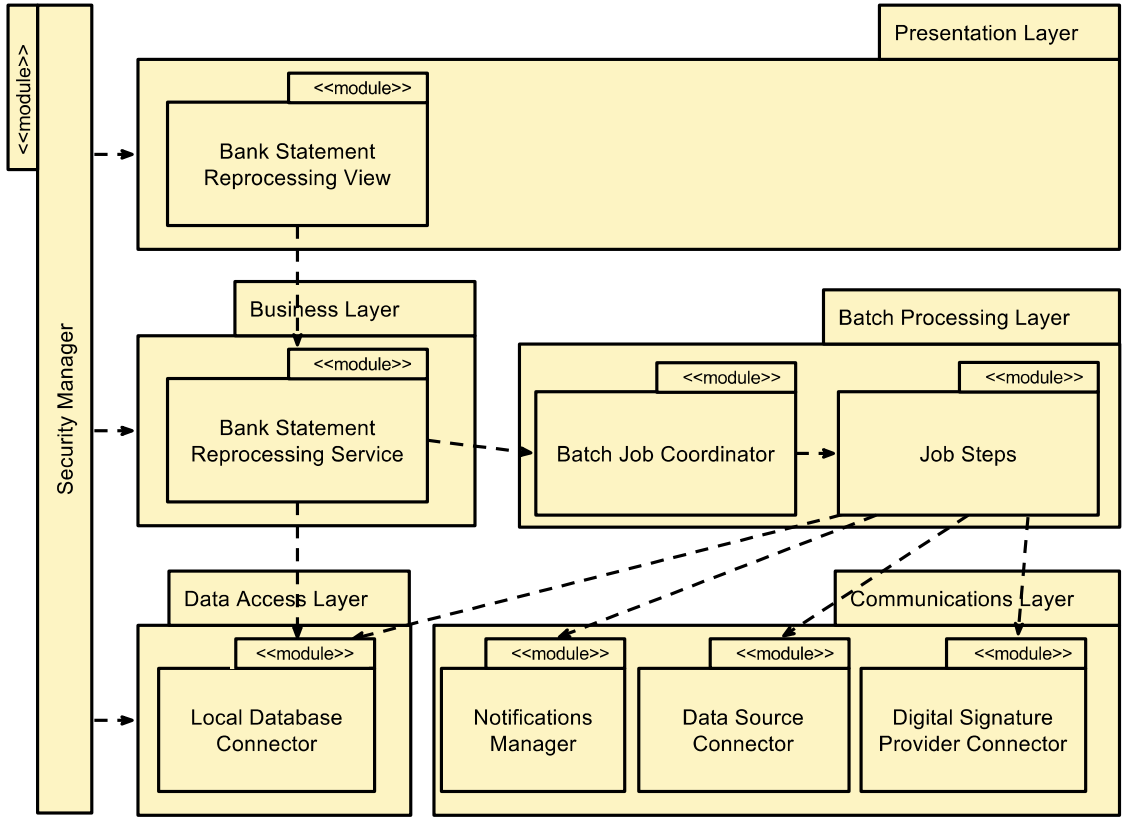
FIGURE 2–3 THREE-TIER CLIENT/SERVER ARCHITECTURE FOR AN AIRLINE BOOKING SYSTEM

Another Example [CK]



[CK, Fig. 6.3]

Yet Another Example [CK]



[CK, Fig. 6.6]

Dynamic Structures [RW]

- The **dynamic structures** of a system define its **runtime** elements and their interactions.
- These internal interactions may be:
  - Flows of information between elements (element A sends messages to element B)
  - The parallel or sequential execution of internal tasks (element X invokes a routine on element Y)
  - Expressed in terms of the effect they have on data (data item D is created, updated many times, and finally destroyed)

Which Structures are Architectural?

- A structure is **architectural** if it supports reasoning about the system and the system's properties.
- The reasoning should be about an attribute of the system that is important to some stakeholder.
  - Link to requirements!
- These include:
  - Functionality achieved by the system
  - The availability of the system in the face of faults
  - The difficulty of making specific changes to the system (modifiability)
  - The responsiveness of the system to user requests,
  - Etc.

Architecture is an Abstraction

- Architecture is about software elements and how they relate to each other.
  - Omits certain information about elements that is not useful for reasoning about the system.
  - It omits information that has **no ramifications outside of a single element**.
    - Private details of elements—details having to do solely with internal implementation—are not architectural.
- The architectural abstraction lets us look at the system in terms of its elements, how they are arranged, how they interact, how they are composed, what their properties are that support our system reasoning, and so forth.
- This **abstraction is essential to taming the complexity of architecture**.
- Every system has a software architecture!
  - Every system has elements and relationships among them
  - It may not be known to anyone

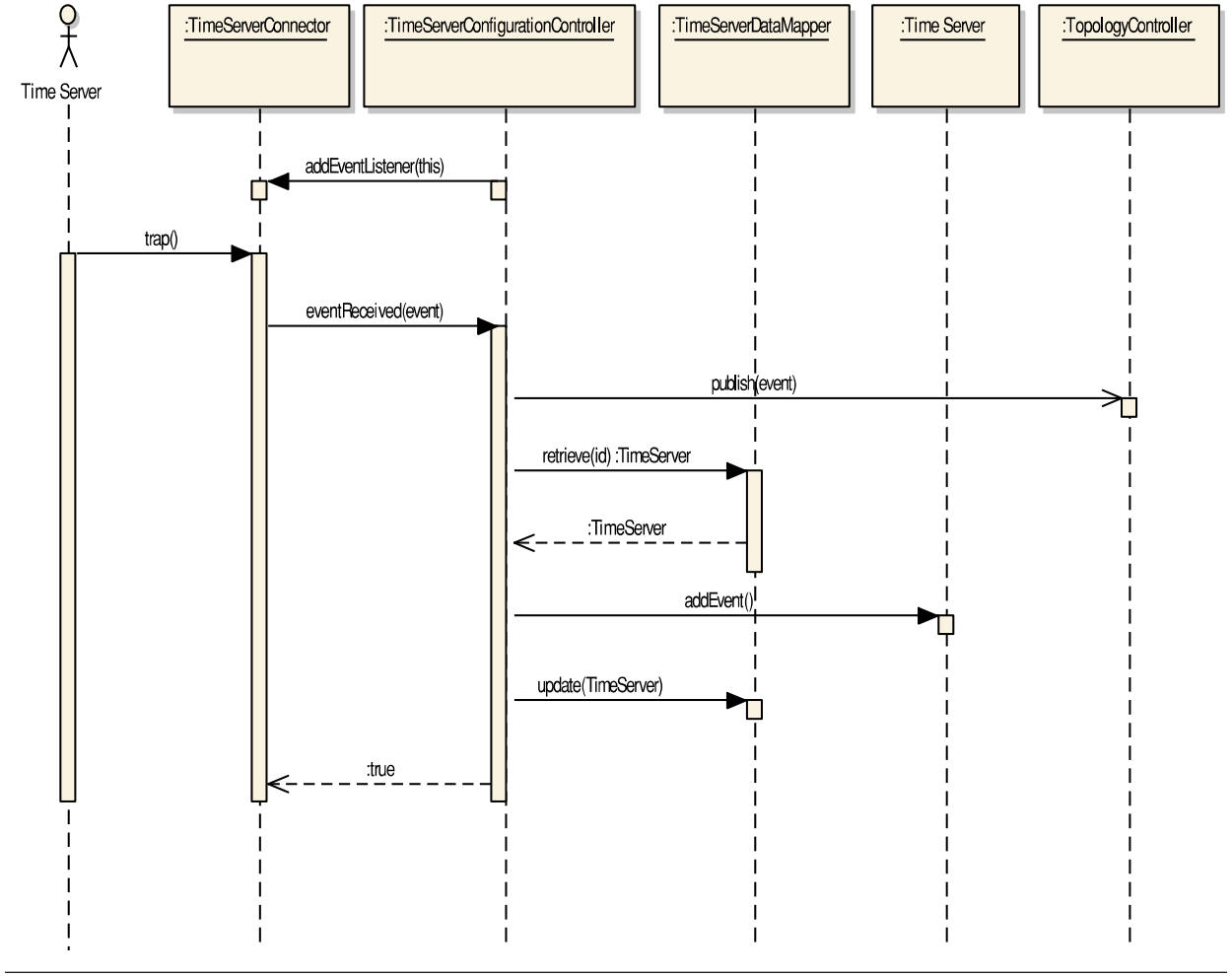
Module Structures (Static)

- **Module structures** embody decisions as to how the system is to be structured as a set of code or data units that have to be constructed or procured.
- In any module structure, the elements are modules of some kind (perhaps classes, or layers, or merely divisions of functionality, all of which are units of implementation).
- Modules are assigned **areas of functional responsibility**; there is less emphasis in these structures on how the software manifests at runtime.
- Module structures allow us to answer questions such as these:
  - What is the primary functional responsibility assigned to each module?
  - What other software elements is a module allowed to use?
  - What other software does it actually use and depend on?
  - What modules are related to other modules by generalization or specialization (i.e., inheritance) relationships?

Component-and-connector Structures (Dynamic)

- **Component-and-connector structures** embody decisions as to how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors).
- Elements are runtime components: services, peers, clients, servers, filters, etc.
- Connectors are the communication vehicles among components: call-return, process synchronization operators, pipes, etc.
- Component-and-connector views help us answer questions such as these:
  - What are the major executing components and how do they interact at runtime?
  - What are the major shared data stores?
  - Which parts of the system are replicated?
  - How does data progress through the system?
  - What parts of the system can run in parallel?
  - Can the system's structure change as it executes and, if so, how?
- These views – crucially important for asking questions about the system's runtime properties such as performance, security, availability, and more.

Example: Sequence Diagram



[CK, Fig. 4.11]

Allocation structures

- **Allocation structures** show the relationship between the software elements and elements in one or more external environments in which the software is created and executed.
- Allocation views help us answer questions such as these:
  - What processor does each software element execute on?
  - In what directories or files is each element stored during development, testing, and system building?
  - What is the assignment of each software element to development teams?

Structures Provide Insight

- Each structure provides a perspective for reasoning about some of the relevant quality attributes.
- For example:
  - The module structure, which embodies what modules use what other modules, is strongly tied to the ease with which a system can be extended or contracted. (Modifiability)
  - The concurrency structure, which embodies parallelism within the system, is strongly tied to the ease with which a system can be made free of deadlock and performance bottlenecks. (Performance)
  - The deployment structure is strongly tied to the achievement of performance, availability, and security goals.
  - And so forth.

Some Useful Module Structures

Decomposition structure

- The units are modules that are related to each other by the *is-a-submodule-of* relation.
- It shows how modules are decomposed into smaller modules recursively until the modules are small enough to be easily understood.
- Modules often have products (such as interface specifications, code, test plans, etc.) associated with them.
- The decomposition structure determines, to a large degree, the system's modifiability, by assuring that likely changes are localized.
- This structure is often used as the basis for the development project's organization, including the structure of the documentation, and the project's integration and test plans.
- The units in this structure tend to have names that are organization-specific such as "segment" or "subsystem."

Some Useful Module Structures

Uses structure.

- The units here are also modules, perhaps classes.
- The units are related by the *uses* relation, a specialized form of dependency.
- A unit of software *uses* another if the correctness of the first requires the presence of a correctly functioning version (as opposed to a stub) of the second.
- The uses structure is used to engineer systems that can be extended to add functionality, or from which useful functional subsets can be extracted.
- The ability to easily create a subset of a system allows for incremental development.

Some Useful Module Structures

Layer structure

- The modules in this structure are called *layers*.
- A layer is an abstract "virtual machine" that provides a cohesive set of services through a managed interface.
- Layers are *allowed to use* other layers in a strictly managed fashion.
  - In strictly layered systems, a layer is only allowed to use a single other layer.
- This structure is imbues a system with portability, the ability to change the underlying computing platform.

Some Useful Module Structures

Class (or generalization) structure

- The module units in this structure are called *classes*.
- The relation is *inherits from* or *is an instance of*.
- This view supports reasoning about collections of similar behavior or capability
  - E.g., the classes that other classes inherit from and parameterized differences
- The class structure allows one to reason about reuse and the incremental addition of functionality.
- If any documentation exists for a project that has followed an object-oriented analysis and design process, it is typically this structure.

Some Useful Module Structures

Data model

- The data model describes the static information structure in terms of data entities and their relationships.
  - For example, in a banking system, entities will typically include Account, Customer, and Loan.
  - Account has several attributes, such as account number, type (savings or checking), status, and current balance.

Some Useful C&C Structures

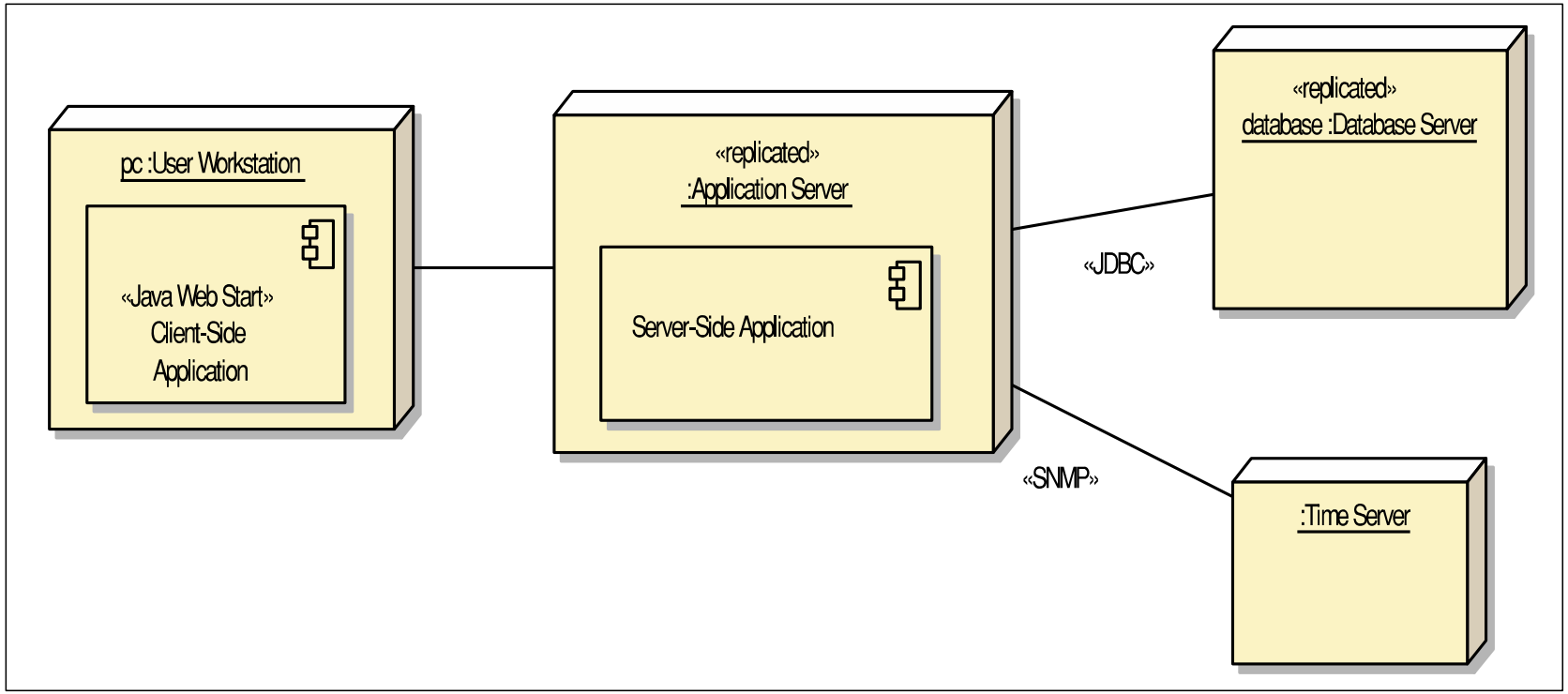
- The relation in all component-and-connector structures is attachment, showing how the components and the connectors are hooked together.
- The connectors can be familiar constructs such as “invokes.”
- Useful C&C structures include:
  - **Service structure**
    - The units are services that interoperate with each other by service coordination mechanisms such as SOAP.
    - The service structure helps to engineer a system composed of components that may have been developed anonymously and independently of each other.
  - **Concurrency structure**
    - This structure helps determine opportunities for parallelism and the locations where resource contention may occur.
    - The units are components
    - The connectors are their communication mechanisms.
    - The components are arranged into logical threads.

Some Useful Allocation Structures

Deployment structure

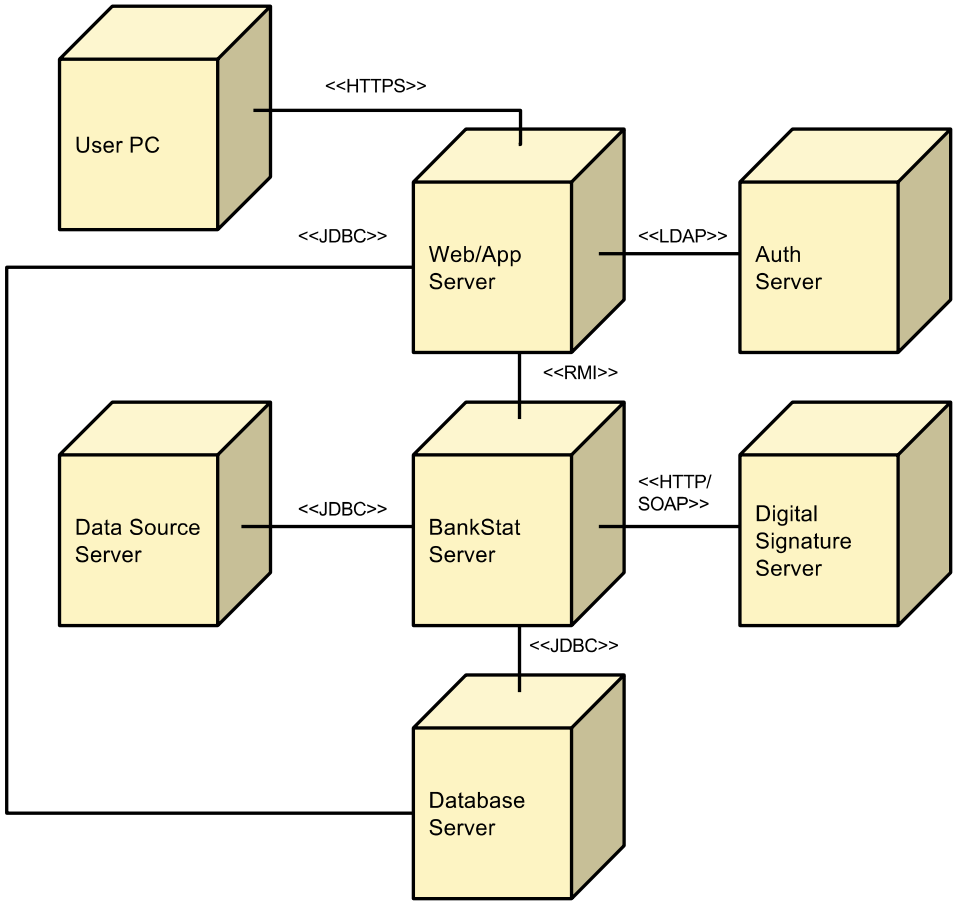
- The deployment structure shows how software is assigned to hardware processing and communication elements.
- The elements are software elements (usually a process from a C&C view), hardware entities (processors), and communication pathways.
- Relations are allocated-to, showing on which physical units the software elements reside, and migrates-to if the allocation is dynamic.
- This structure can be used to reason about performance, data integrity, security, and availability.
- It is of particular interest in distributed and parallel systems.

Example: Deployment



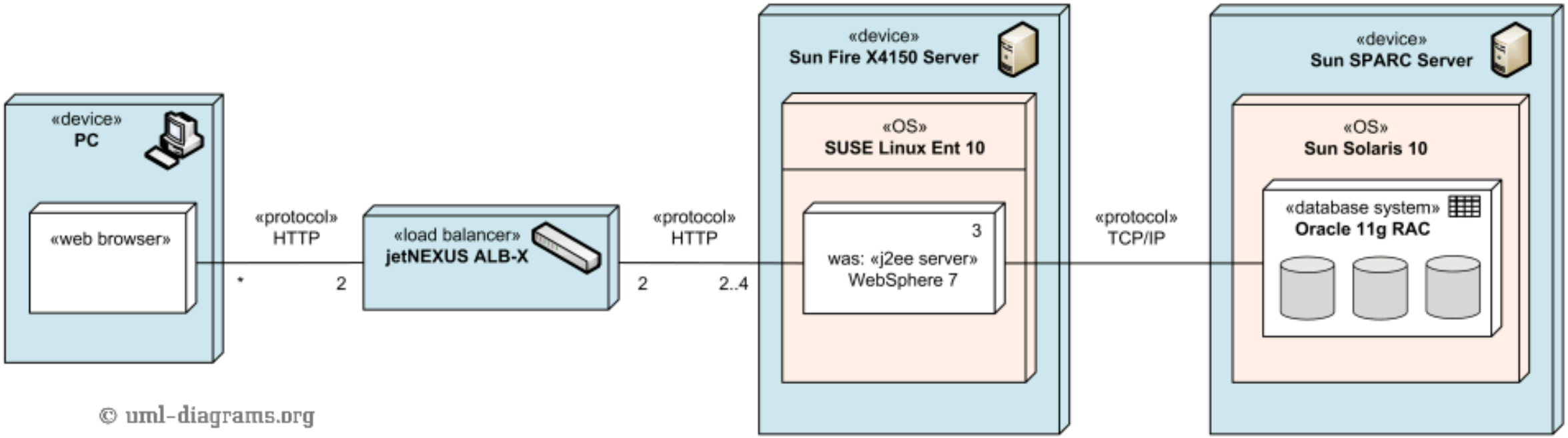
[CK, Fig. 4.4]

Another Example: Deployment

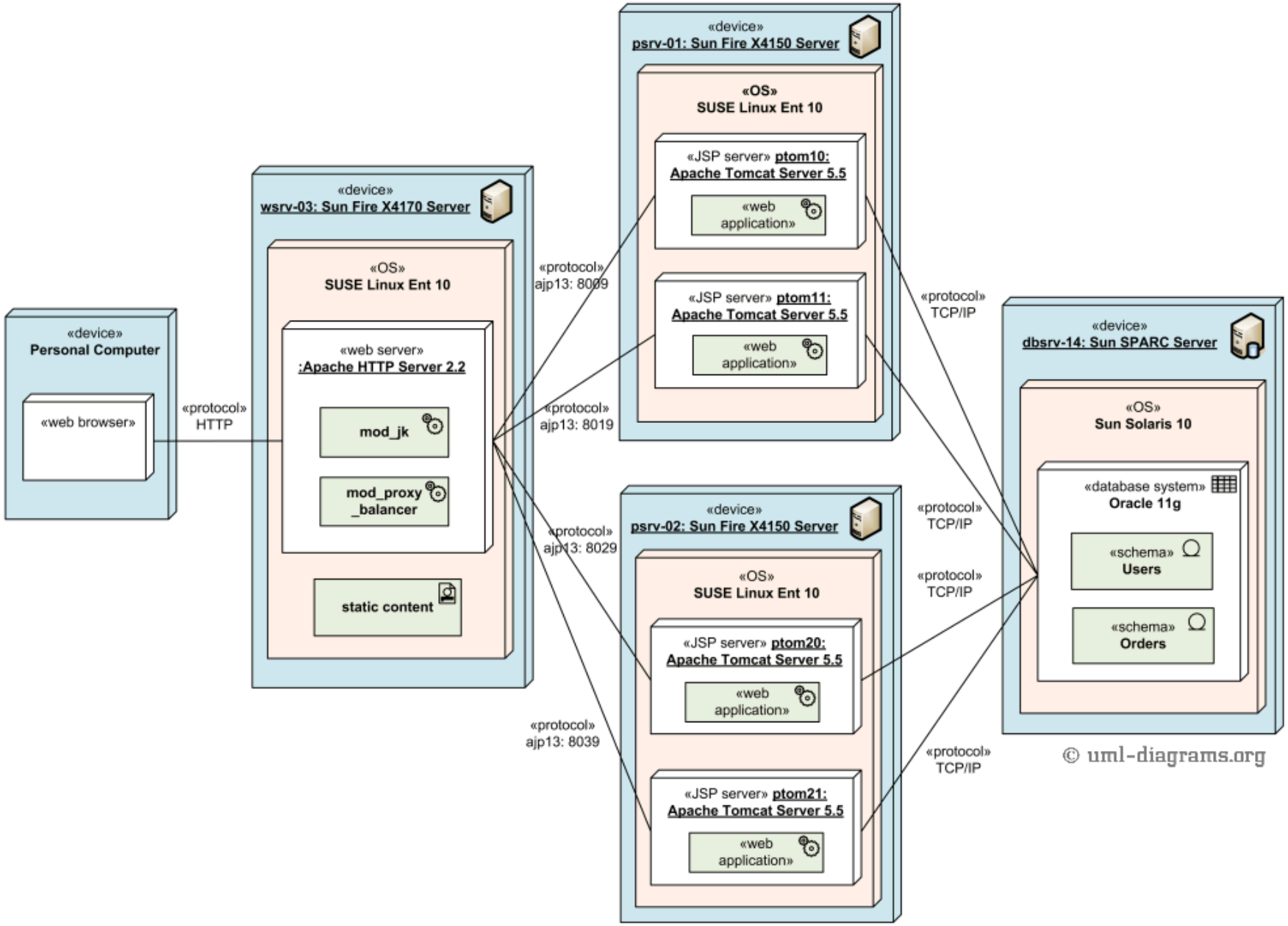


[CK, Fig. 6.5]

Yet Another Deployment Example



Last One...


«device»
Personal Computer
«web browser»
«protocol»
HTTP
«device»
wsrv-03: Sun Fire X4170 Server
«OS»
SUSE Linux Ent 10
«web server»
:Apache HTTP Server 2.2
mod_jk
mod_proxy
_balancer
static content
«protocol»
ajp13: 8009
«protocol»
ajp13: 8019
«protocol»
ajp13: 8029
«protocol»
ajp13: 8039
«device»
psrv-01: Sun Fire X4150 Server
«OS»
SUSE Linux Ent 10
«JSP server» ptom10:
Apache Tomcat Server 5.5
«web application»
«JSP server» ptom11:
Apache Tomcat Server 5.5
«web application»
«device»
psrv-02: Sun Fire X4150 Server
«OS»
SUSE Linux Ent 10
«JSP server» ptom20:
Apache Tomcat Server 5.5
«web application»
«JSP server» ptom21:
Apache Tomcat Server 5.5
«web application»
«protocol»
TCP/IP
«protocol»
TCP/IP
«protocol»
TCP/IP
«protocol»
TCP/IP
«device»
dbsrv-14: Sun SPARC Server
«OS»
Sun Solaris 10
«database system»
Oracle 11g
«schema»
Users
«schema»
Orders
© uml-diagrams.org

Some Useful Allocation Structures

Implementation structure

- This structure shows how software elements (usually modules) are mapped to the file structure(s) in the system's development, integration, or configuration control environments.
- This is critical for the management of development activities and build processes.

Work assignment structure

- This structure assigns responsibility for implementing and integrating the modules to the teams who will carry it out.
- Having a work assignment structure be part of the architecture makes it clear that the decision about who does the work has architectural as well as management implications.
- The architect will know the expertise required on each team.
- This structure will also determine the major communication pathways among the teams: regular teleconferences, wikis, email lists, and so forth.

Relating Structures to Each Other

- Elements of one structure will be related to elements of other structures, and we need to reason about these relations.
  - A module in a decomposition structure may be manifested as one, part of one, or several components in one of the component-and-connector structures.
- In general, mappings between structures are many to many.

Modules vs. Components

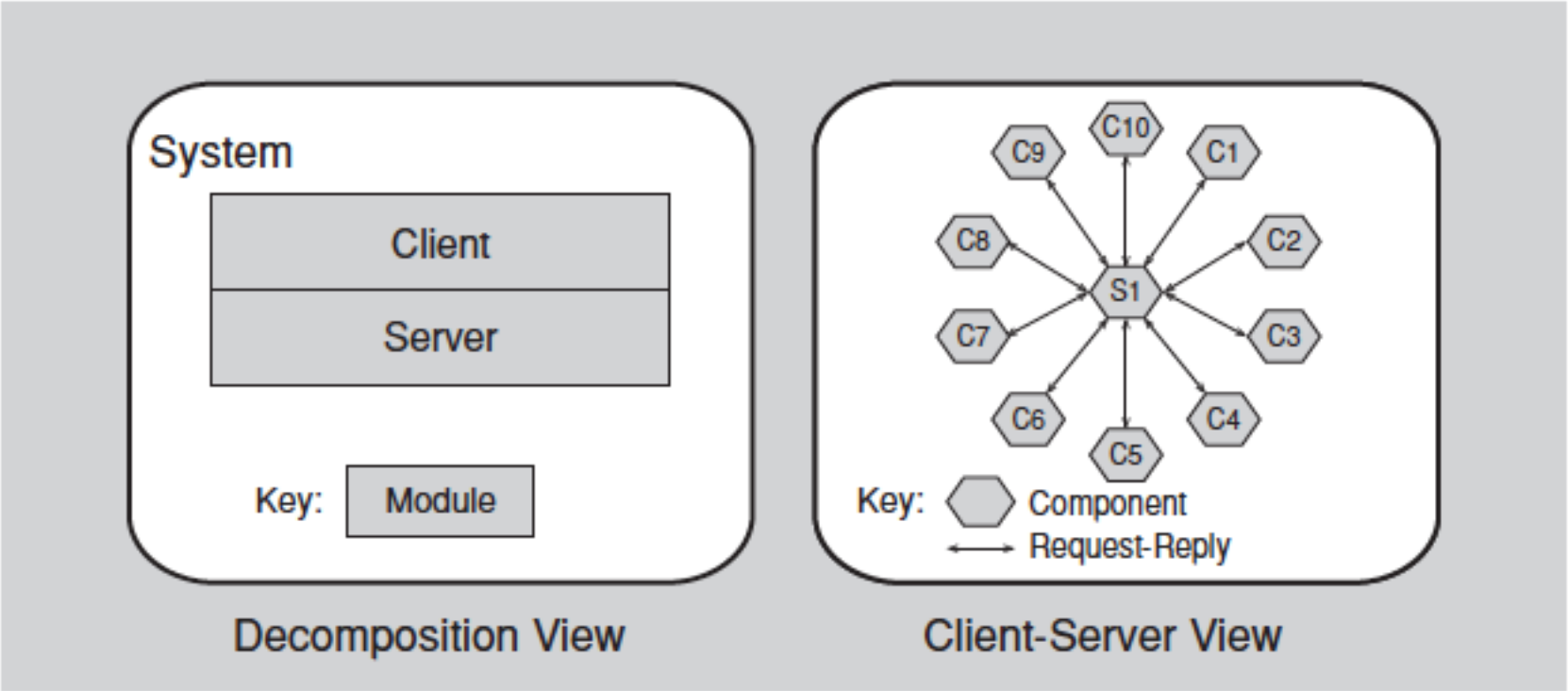


FIGURE 1.2 Two views of a client-server system

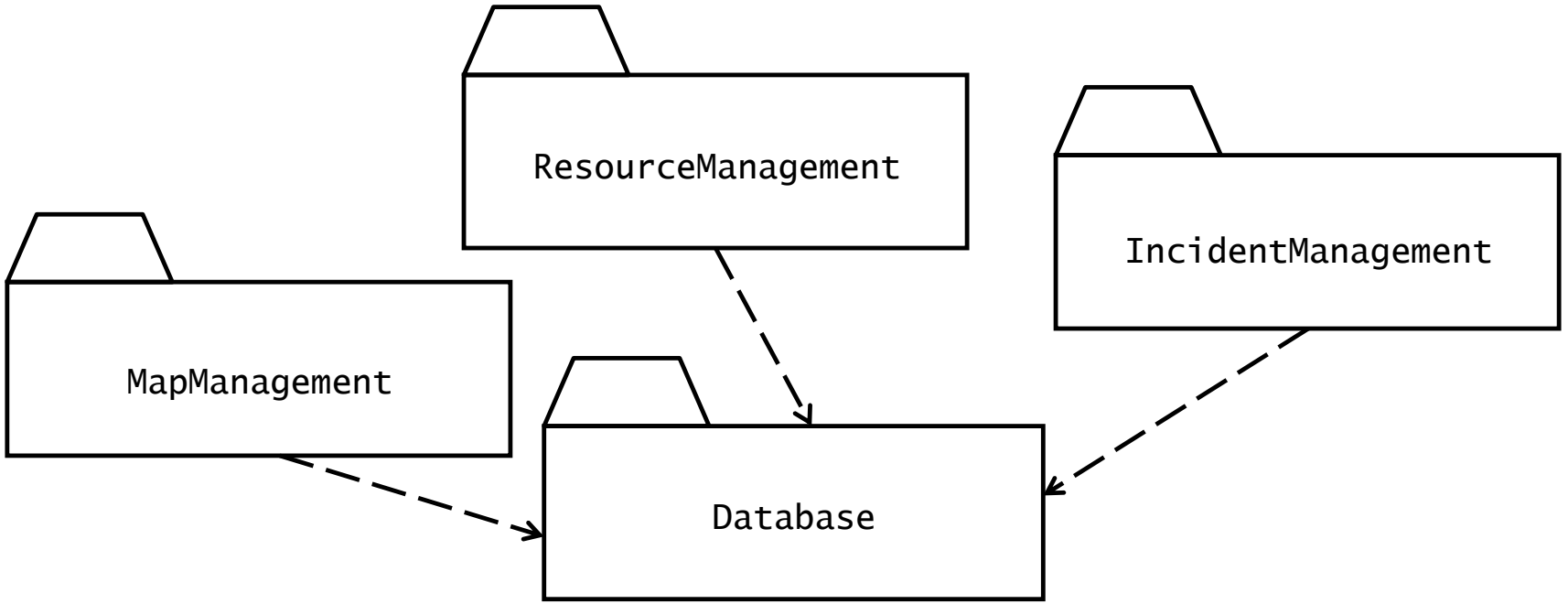
Cohesion and Coupling at the Architectural Level

Design Principles: Coupling and Cohesion

- Goal: Reduce **complexity while allowing change**
- **Cohesion** measures the dependencies among elements of (sub)systems
  - High cohesion: The elements in the subsystem perform similar tasks and are related to each other (via associations)
  - Low cohesion: Lots of miscellaneous and auxiliary elements, no associations
- **Coupling** measures dependencies between subsystems
  - High coupling: Changes to one subsystem will have high impact on the other subsystem (change of model, massive recompilation, etc.)
  - Low coupling: A change in one subsystem does not affect any other subsystem

Example Of Reducing The Coupling Of Subsystems

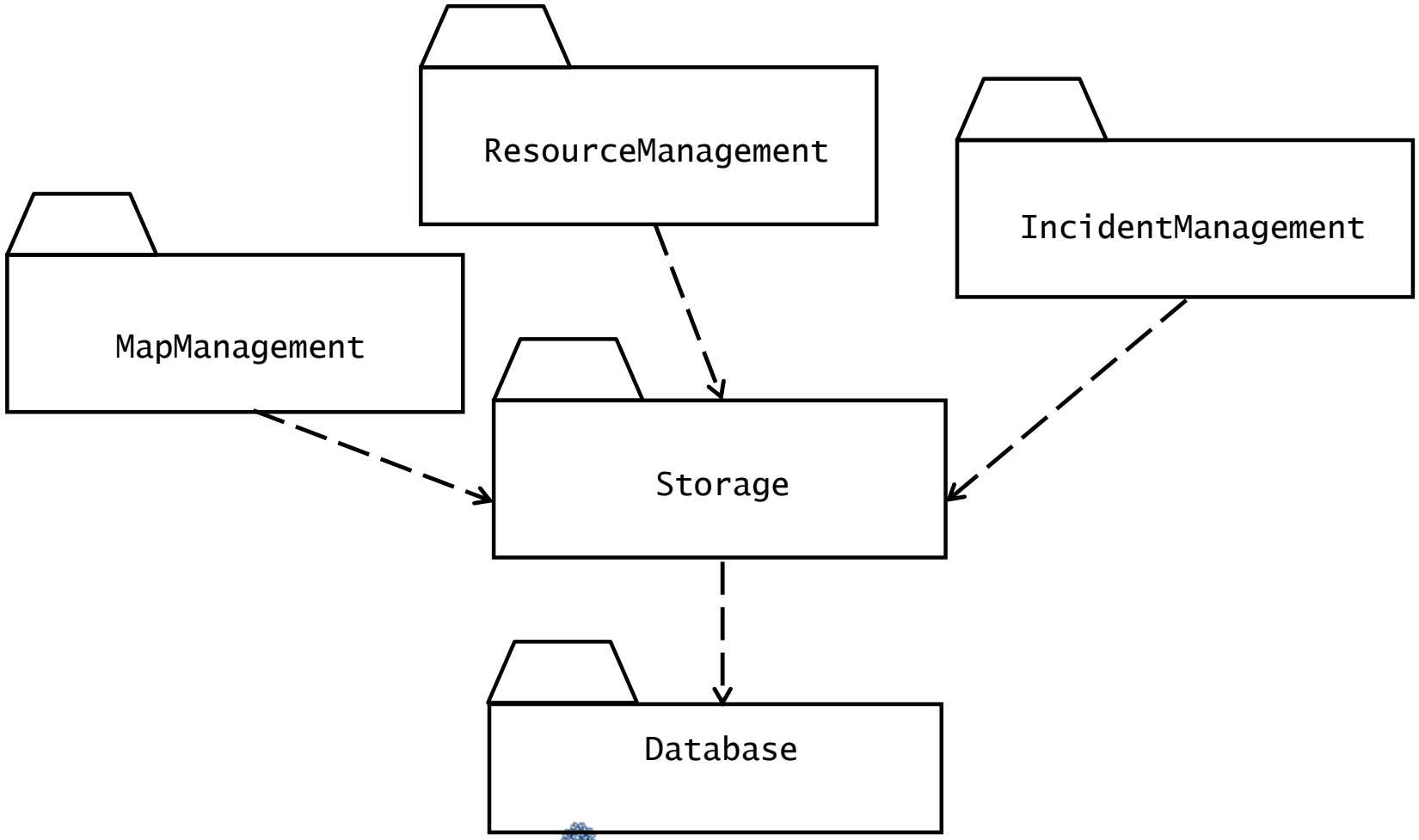
- Alternative 1: Direct access to the Database subsystem
  - Note: UML package diagrams are used here



[Bruegge and Dutoit, OO Software Engineering Using UML, Patterns, and Java, 3rd Ed., 2010]

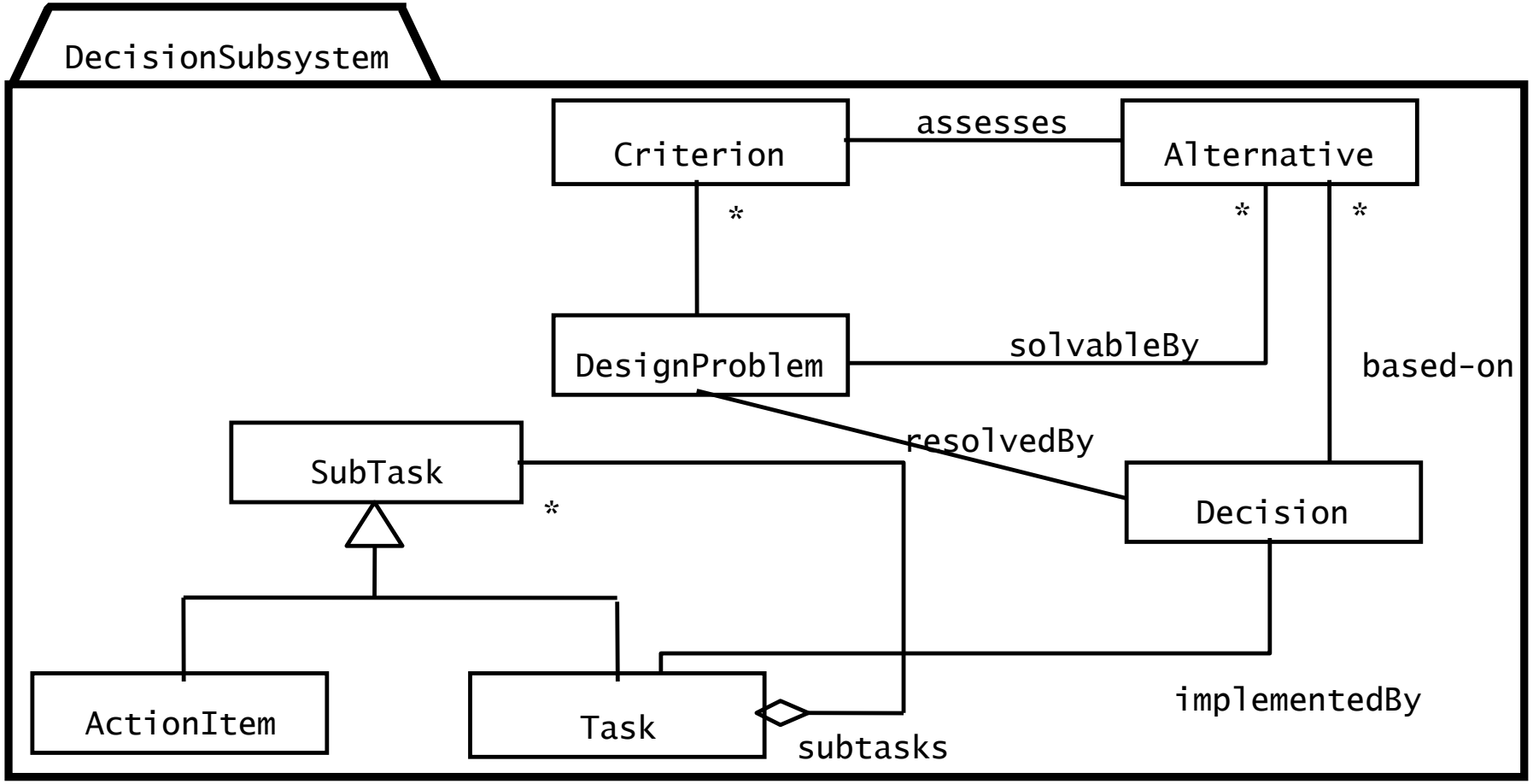
Example Of Reducing The Coupling Of Subsystems

- Alternative 2: Indirect access to the Database through a Storage subsystem



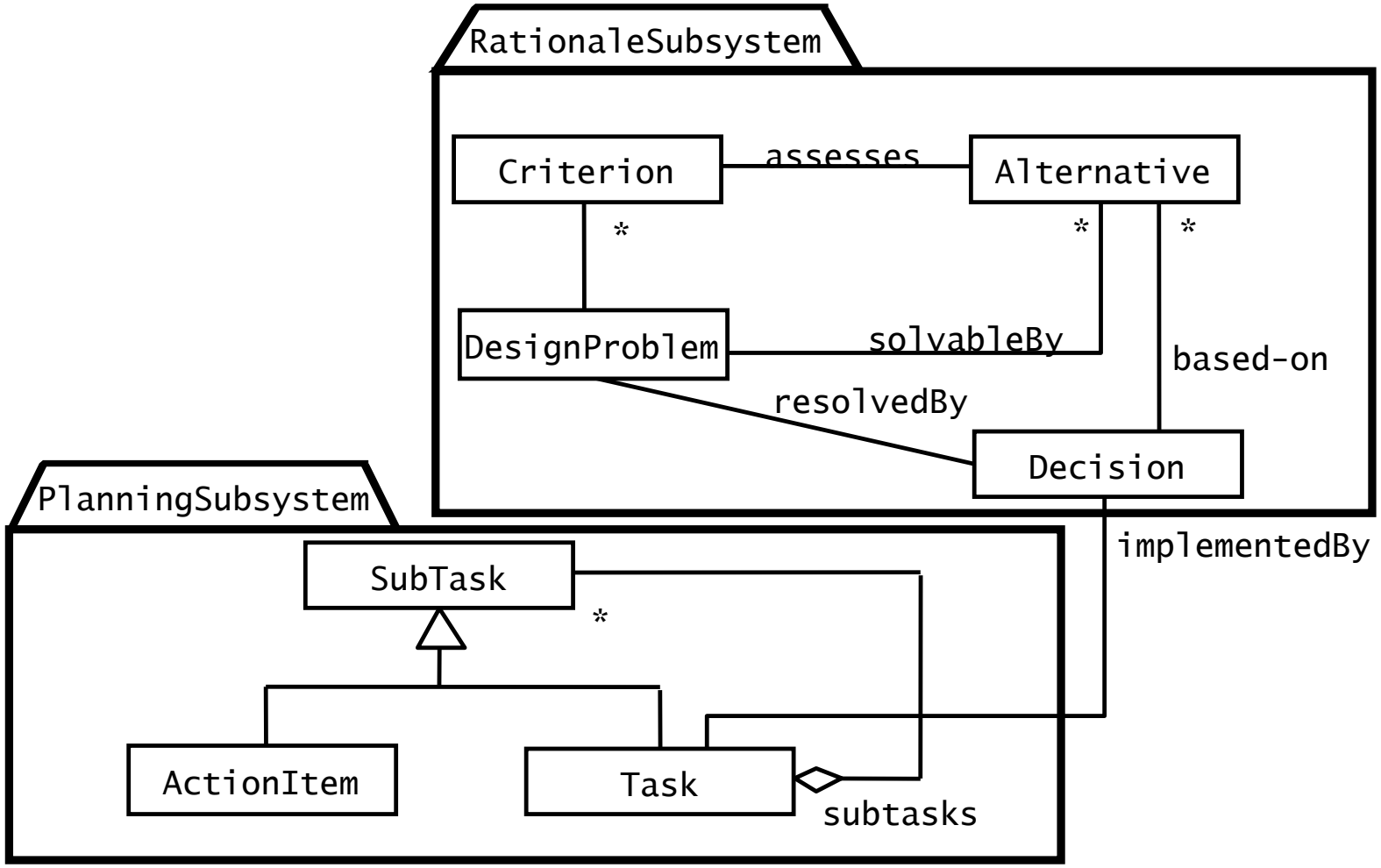
[Bruegge and Dutoit, OO Software Engineering Using UML, Patterns, and Java, 3rd Ed., 2010]

Design Decision Tracking System (Low Cohesion)



[Bruegge and Dutoit, OO Software Engineering Using UML, Patterns, and Java, 3rd Ed., 2010]

Alternative Subsystem Decomposition



[Bruegge and Dutoit, OO Software Engineering Using UML, Patterns, and Java, 3rd Ed., 2010]

Examples of Architectural Styles/Patterns

Architectural Patterns [1]

- Architectural elements can be composed in ways that solve particular problems.
  - The compositions have been found useful over time, and over many different domains
  - They have been documented and disseminated.
  - These compositions of architectural elements, called **architectural patterns**.
  - Patterns provide packaged strategies for solving some of the problems facing a system.
- E.g., a common module type pattern is the Layered pattern.
  - When the *uses* relation among software elements is strictly unidirectional, a system of layers emerges.
  - A layer is a coherent set of related functionality.
  - Many variations of this pattern, lessening the structural restriction, occur in practice.

Architectural Patterns [2]

Common component-and-connector type patterns:

- Shared-data (or repository) pattern
  - This pattern comprises components and connectors that create, store, and access persistent data.
  - The repository usually takes the form of a (commercial) database.
  - The connectors are protocols for managing the data, such as SQL.
- Client-server pattern
  - The components are the clients and the servers.
  - The connectors are protocols and messages they share among each other to carry out the system's work.

Architectural Patterns [3]

Common allocation patterns:

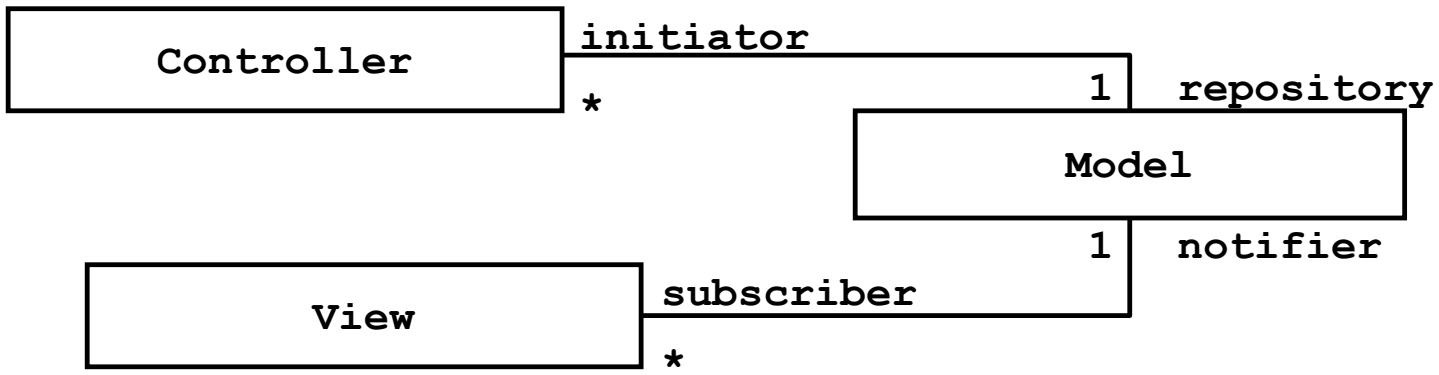
- Multi-tier pattern
  - Describes how to distribute and allocate the components of a system in distinct subsets of hardware and software, connected by some communication medium.
  - This pattern specializes the generic deployment (software-to-hardware allocation) structure.
- **Competence center** pattern and **platform** pattern
  - These patterns specialize a software system's work assignment structure.
  - In competence center, work is allocated to sites depending on the technical or domain expertise located at a site.
  - In platform, one site is tasked with developing reusable core assets of a software product line, and other sites develop applications that use the core assets.
- Others: Layered, Model-View-Controller (MVC), Peer-to-peer, etc.

Example: Model-View-Controller

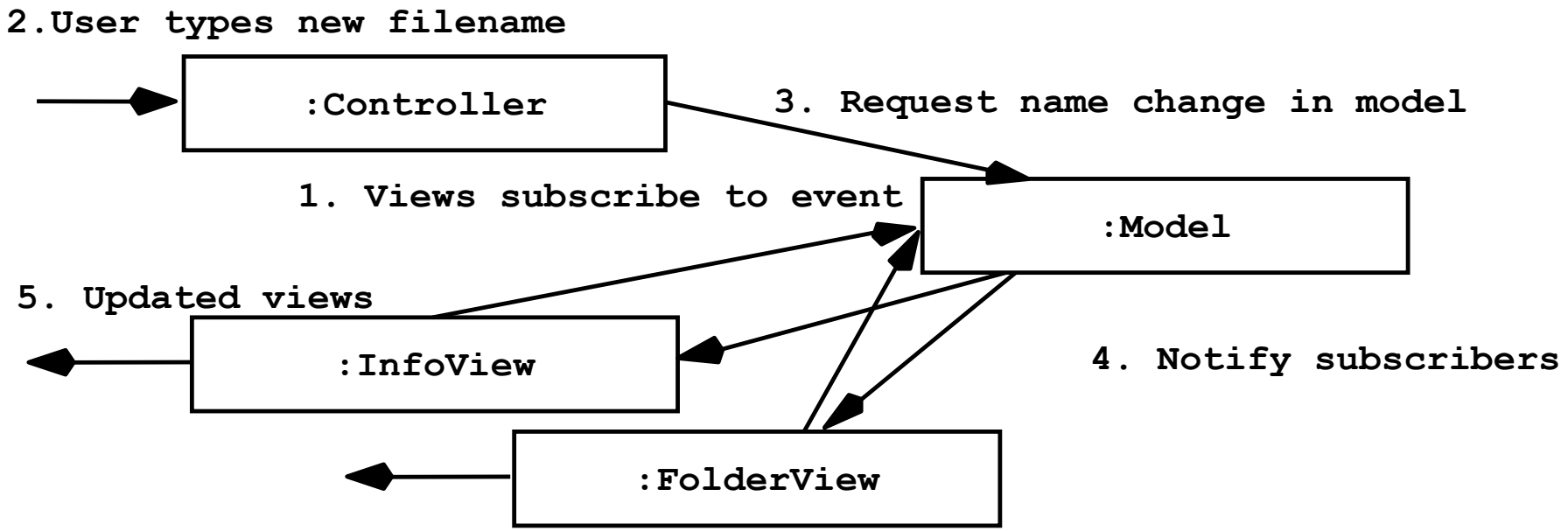
- A decomposition of an interactive system into three components:
  - A **model** containing the core functionality and data,
  - One or more **views** displaying information to the user, and
  - One or more **controllers** that handle user input
- A change-propagation mechanism (i.e., subscribe/notify protocol) ensures consistency between user interface and model, e.g.,
  - If the user changes the model through the controller of one view, the other views will be updated automatically
- Sometimes the need for the controller to operate in the context of a given view may mandate combining the view and the controller into one component
- The division into the MVC components improves maintainability

Model/View/Controller

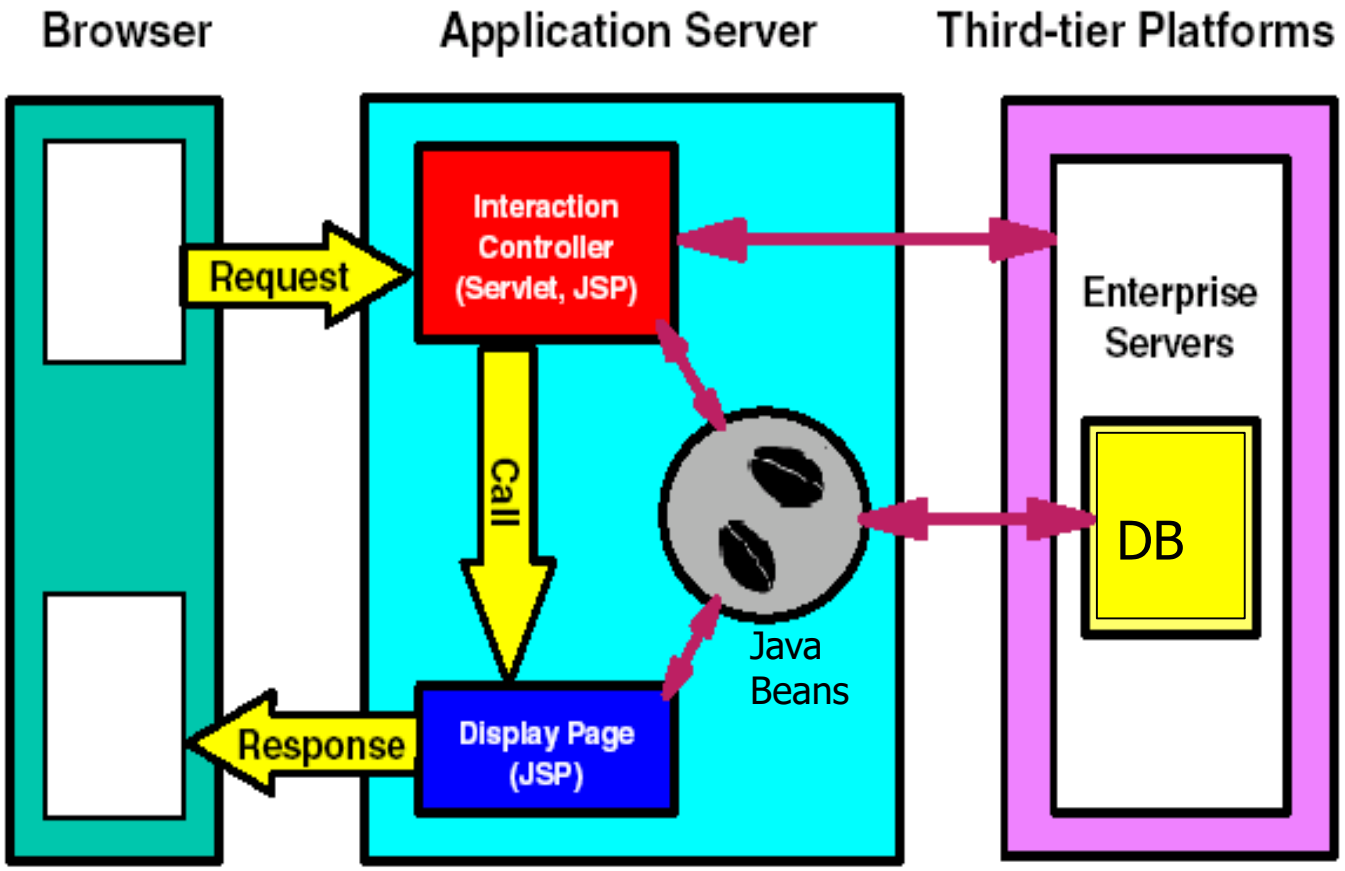
- Subsystems are classified into 3 different types
  - Model subsystem: **Responsible for application domain knowledge**
  - View subsystem: **Responsible for displaying application domain objects to the user**
  - Controller subsystem: **Responsible for sequence of interactions with the user and notifying views of changes in the model.**
- MVC is a special case of a repository architecture:
  - Model subsystem implements the central data structure, the Controller subsystem explicitly dictate the control flow



Example: UML Collaboration Diagram for MVC



Model View Controller Architecture for Web Applications

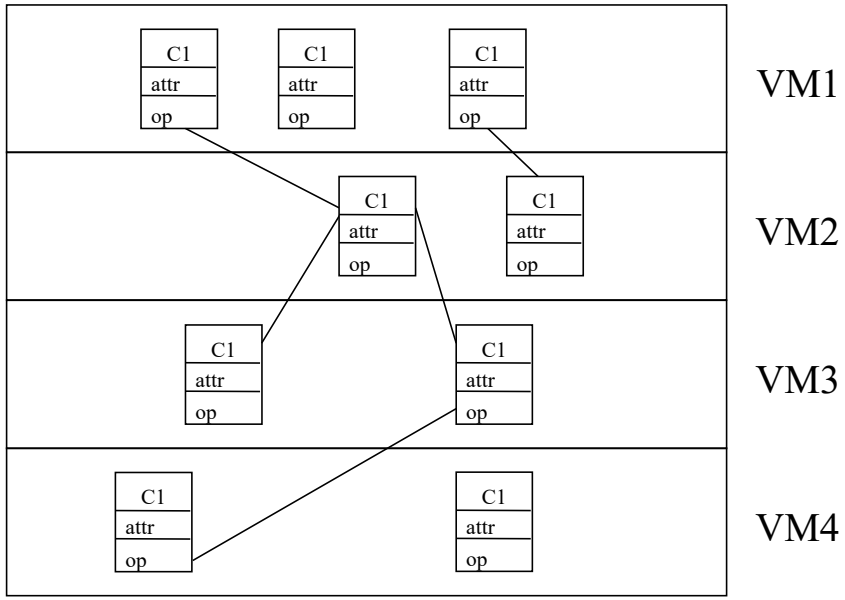


Layered Systems

- "A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below." (Garlan and Shaw)
- Each layer collects services at a particular level of abstraction
- Layers are sometimes called virtual machines (VMs)
- In a pure layered system: Layers are hidden to all except adjacent layers

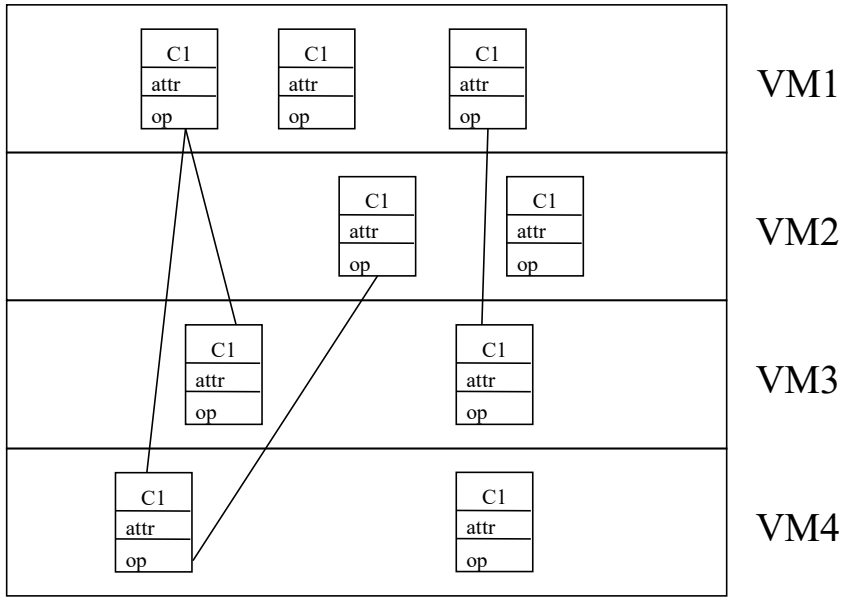
Closed Architecture (Opaque Layering)

- Any layer can only invoke operations from the immediate layer below
- Design goal:
  High maintainability, flexibility



Open Architecture (Transparent Layering)

- Any layer can invoke operations from any layers below
- Design goal: **Runtime efficiency**



Layered System Examples

- Example 1: ISO defined the OSI 7-layer architectural model with layers: Application, Presentation, …, Data, Physical.
  - Protocol specifies behaviour at each level of abstraction (layer).
  - Each layer deals with specific level of communication and uses services of the next lower level.

- Example 2: TCP/IP is the basic communications protocol used on the internet. The same layers in a network communicate ‘virtually’.

- Example 3: Operating systems e.g. hardware layer, …, kernel, resource management, … user level “Onion Skin model”.

- ...

Layered Systems

- Strengths
  - Increasing levels of abstraction as we move up through layers – partitions complex problems
  - Maintenance - in theory, a layer only interacts with layers above and below. Change has minimum effect.
  - Reuse - different implementations of the same level can be interchanged
  - Standardisation based on layers e.g. OSI
- Weaknesses
  - Not all systems are easily structured in layers (e.g., mobile robotics)
  - Performance - communicating down through layers and back up, hence bypassing may occur for efficiency reasons

Tiered Architectures

- Special kind of layered architecture for enterprise applications
- Evolution
  - Two Tier
  - Three Tier
  - Multi Tier

2-Tier Client/Server Architectural Style

- One or many servers provides services to instances of subsystems, called clients.
- Client calls on the server, which performs some service and returns the result
  - Client knows the *interface* of the server *(its service)*
  - Server does not need to know the interface of the client
- Users interact only with the client

Client/Server Architectural Style

- Often used in database systems:
  - Front-end: User application (client)
  - Back end: Database access and manipulation (server)
- Functions performed by client:
  - Customized user interface
  - Front-end processing of data
  - Initiation of server remote procedure calls
  - Access to database server across the network
- Functions performed by the database server:
  - Centralized data management
  - Data integrity and database consistency
  - Database security
  - Concurrent operations (multiple user access)
  - Centralized processing (for example archiving)

Design Goals for Client/Server Systems

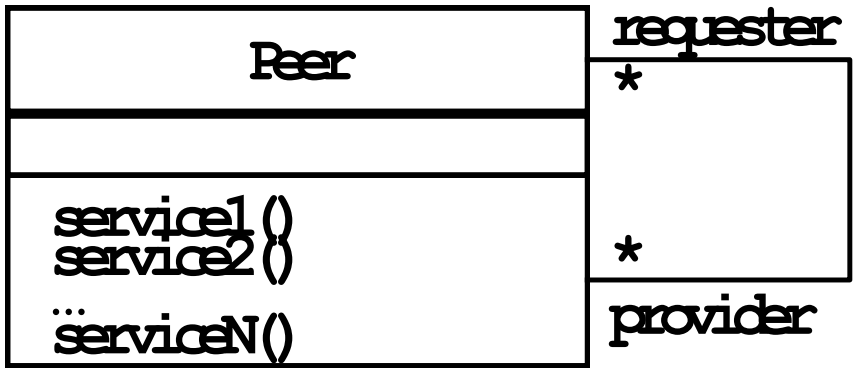
- *Service Portability*
  - Server can be installed on a variety of machines and operating systems and functions in a variety of networking environments
- *Transparency, Location-Transparency*
  - The server might itself be distributed (why?), but should provide a single "logical" service to the user
- *Performance*
  - Client should be customized for interactive display-intensive tasks
  - Server should provide CPU-intensive operations
- *Scalability*
  - Server should have spare capacity to handle larger number of clients
- *Flexibility*
  - The system should be usable for a variety of user interfaces and end devices (eg. wearable computer, desktop, cell, PDA)
- *Reliability*
  - System should survive node or communication link problems

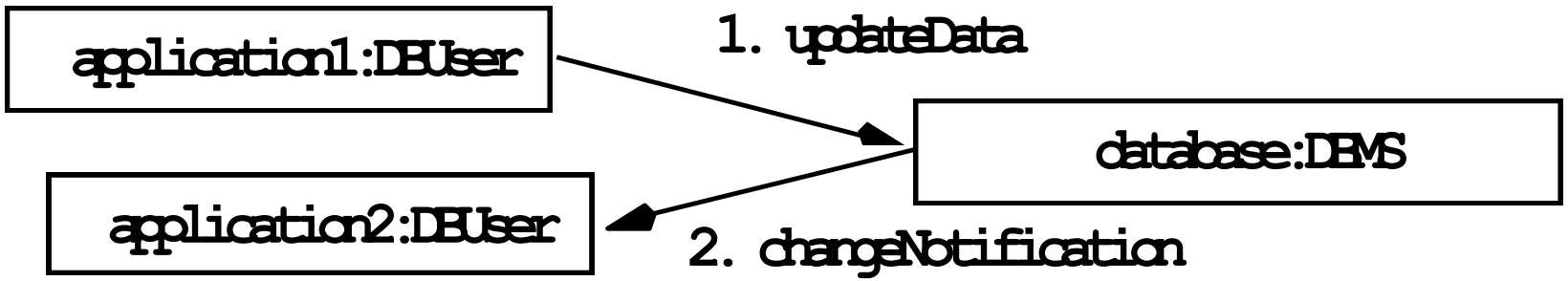
Problems with Client/Server Architectural Styles

- Layered systems do not provide peer-to-peer communication
- Peer-to-peer communication is often needed
- Example: Database receives queries from application but also sends notifications to application when data has changed

Peer-to-Peer Architectural Style

- Generalization of Client/Server Architecture
- Clients can be servers and servers can be clients
- More difficult because of possibility of deadlocks
- Examples??





Three-Tier Client/Server Architecture Design

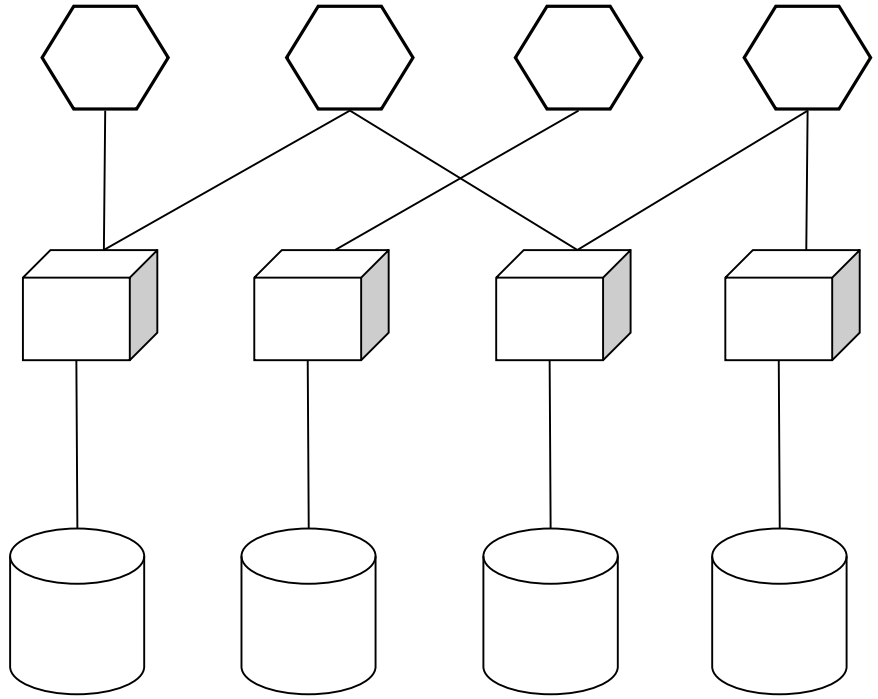
- Emerged in the 1990s to overcome the limitations of the two tier architecture by adding an additional middle tier
- This middle tier provides process management where business logic and rules are executed and can accommodate hundreds of users by providing generic services such as queuing, application execution, and database staging
- An effective distributed client/server design that provides increased performance, flexibility, maintainability, reusability, and scalability, while hiding the complexity of distributed processing from the user

Three Tier Client Server Architecture Design

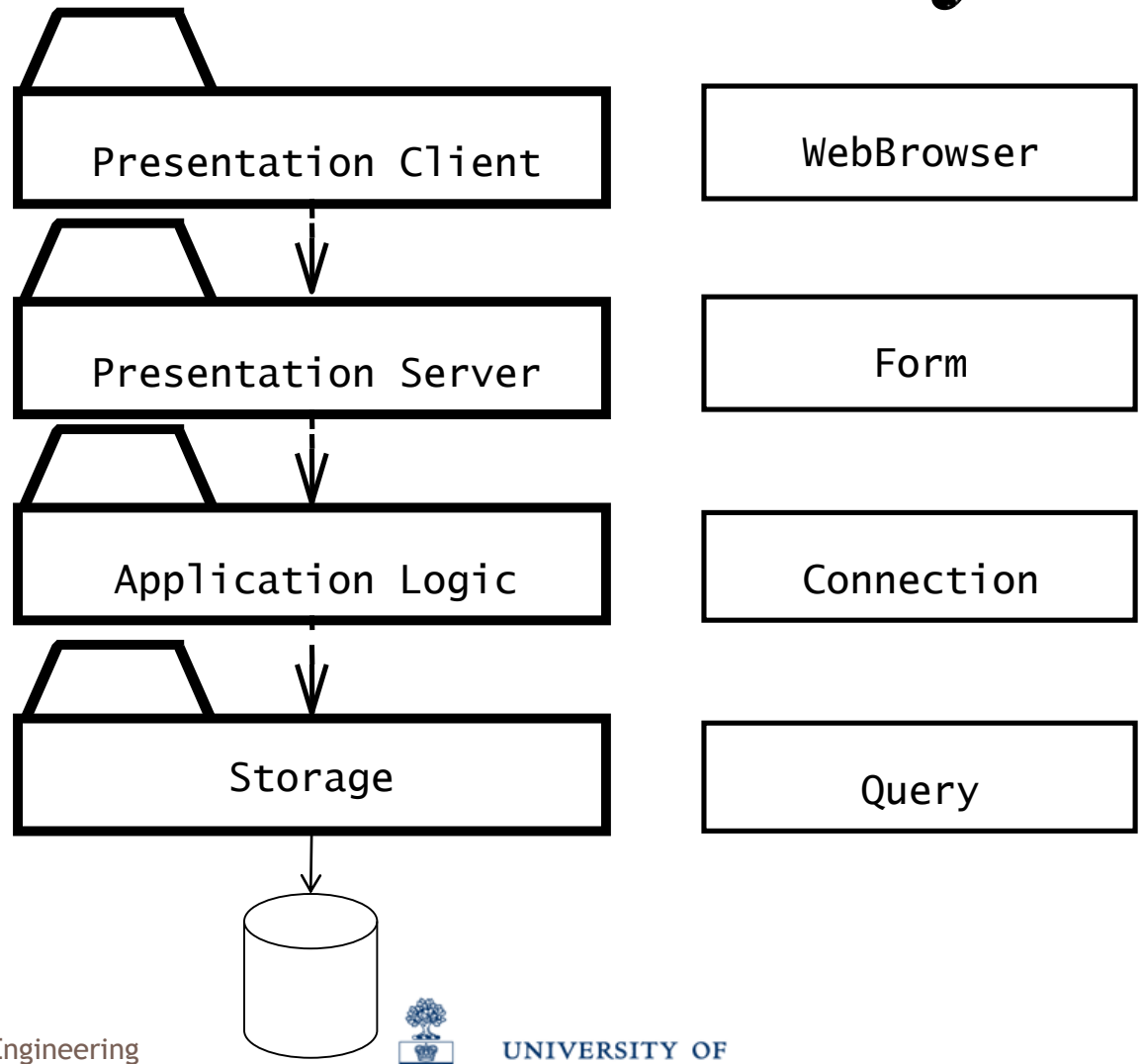
Presentation Logic

Business Logic

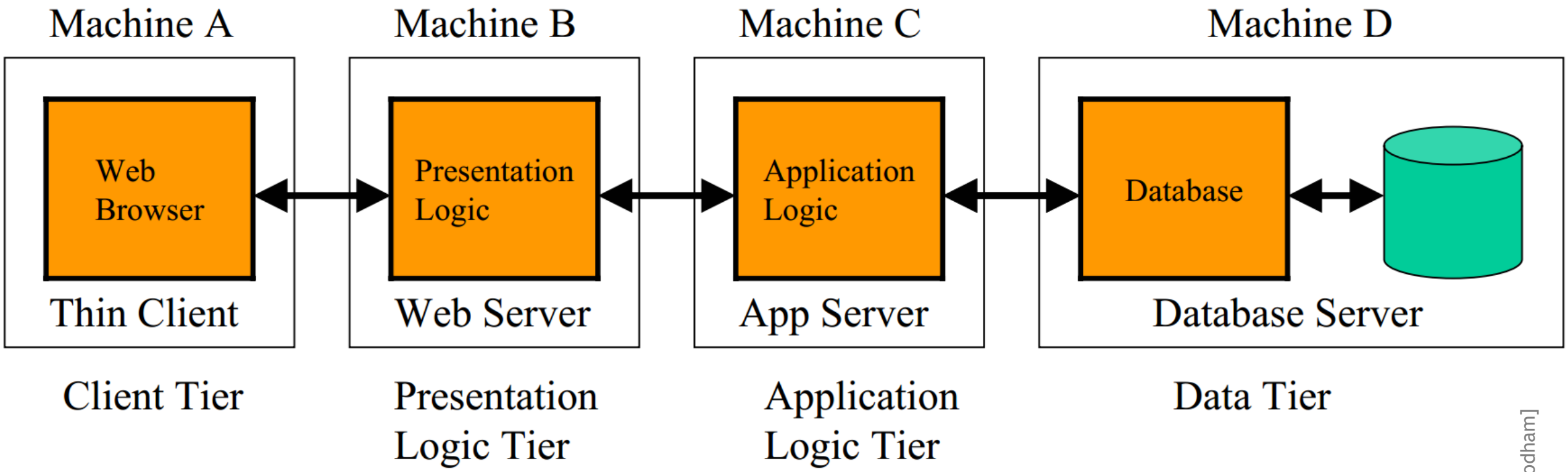
Data



4 tier architectural style.



4 tier architectural style.



[Ken Rodham]

Heterogeneous Architectures

- In practice the architecture of large-scale system is a combination of architectural styles:
  - 'Hierarchical heterogeneous' A Component in one style may have an internal style developed in a completely different style (e.g., pipe component developed in OO style, implicit invocation module with a layered internal structure, etc.)
  - 'Locational heterogeneous' Overall architecture at same level is a combination of different styles (e.g., repository (database) and main program-subroutine, etc.)
    Here individual components may connect using a mixture of architectural connectors - message invocation and implicit invocation.
  - 'Perspective heterogeneous' Different architecture in different perspectives (e.g., structure of the logical view, structure of the physical view, etc.)

What Makes a "Good" Architecture?

- There is no such thing as an inherently good or bad architecture.
- Architectures are either more or less fit for some purpose
- Architectures can be evaluated but only in the context of specific stated goals.