Software Architecture

Lei Yang
sely@scut.edu.cn

Motivation of this course

- Be able to think on the level of ***software architecture*** in all the phases of software engineering
- Algorithm level
- Programming (language) level

教科书和参考书

- 教科书:
  - 软件构架实践（第3版），L. Bass, P. Clements, and R. Kazman，清华大学出版社(2013)
- 参考书:
  - 软件架构实践（第4版），L. Bass, P. Clements, and R. Kazman，机械工业出版社，2022年
  - 软件体系结构，林荣恒 吴步丹 金芝，中国工信出版集团，人民邮电出版社 2016年

考试与成绩

- 期末考试 70%
- 平时成绩 30%
  - 大作业
  - 课堂考勤

Major Contents

- Introduction of software architecture
  - what, why, contexts of software architecture
- Quality attributes
  - Availability, modifiability, performance, security, testability, and usability
  - Architecture pattern & tactics
  - QA analysis and case study
- Architecture in the life cycle
  - Requirement, design, implementation, test, and evaluation
- Architecture in cloud computing

序号	知识单元/章节	知识点	教学要求	课堂教学学时
1	绪论	1.软件体系结构基本概念 2.软件体系结构的重要性	教学要求：（1）理解软件结构的概念（2）掌握软件结构三种视图（3）理解软件体系结构的重要性及不同上下文中的作用思政元素：	4学时
2	质量属性	1.理解质量属性2.可用性3.互操作性4.可修改性5.性能6.安全性7.可测试性8.易用性9.其他质量属性	教学要求：（1）理解描述质量属性场景的六要素（2）能描述不同质量属性场景（3）理解不同质量属性在系统结构设计中的关系重点：可用性、性能。难点：各个质量属性之间的关系。	16学时
3	软件架构战术与模式	架构模式架构战术与模式之间关系使用架构战术	教学要求：理解软件结构战术和模式的定义掌握常见的软件架构模式与战术重点：常用软件架构模式。难点：软件结构战术与模式之间的关系。	4学时
4	质量属性分析与建模	1.质量属性分析使能-架构建模2.实验、仿真与原型3.案例分析-移动云系统性能建模与分析	教学要求：（1）理解架构在质量属性中发挥的作用（2）理解质量属性分析的几种模型方法，以性能为例着重理解质量属性建模与分析方法。重点：质量属性分析的模型与方法。难点：软件架构与质量属性分析之间关系。	4学时
5	敏捷项目中软件架构	敏捷软件开发与架构之间的关系	教学要求：（1）了解敏捷软件开发的概念及方法；（2）理解敏捷软件开发与软件架构之间的关系	2学时
6	软件架构与需求	需求文档中获取架构重要需求、效能树	教学要求：（1）了解获取架构重要需求的方法（2）掌握使用效能树描述架构重要需求重点：效能树构建架构重要需求 难点：综合运用不同方法获取架构重要需求	3学时
7	软件架构设计	设计策略属性驱动设计方法（ADD）ADD流程和具体步骤	教学要求：（1）了解架构设计的不同策略（2）掌握ADD方法，学会使用ADD设计软件架构重点：属性驱动设计方法步骤难点：架构设计策略	2学时
8	软件架构实现、测试与评估	架构与实现架构与测试架构评估因素轻量级架构评估	教学要求：（1）了解结构实现与测试方法（2）掌握架构评估方法重点：轻量级架构评估难点：架构取舍分析方法	2学时
9	云计算中软件架构	云计算及架构	教学要求：（1）云计算基本概念和技术；（2）云计算性能、可用性、安全的架构保障	3学时

Chapter 1

What is Software Architecture?

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.

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

- 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

- 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*.
- A component is always a runtime entity.
  - In SOA, the system is to be built as a set of services.
  - These services are made up of (compiled from) the programs in the various implementation units – modules.

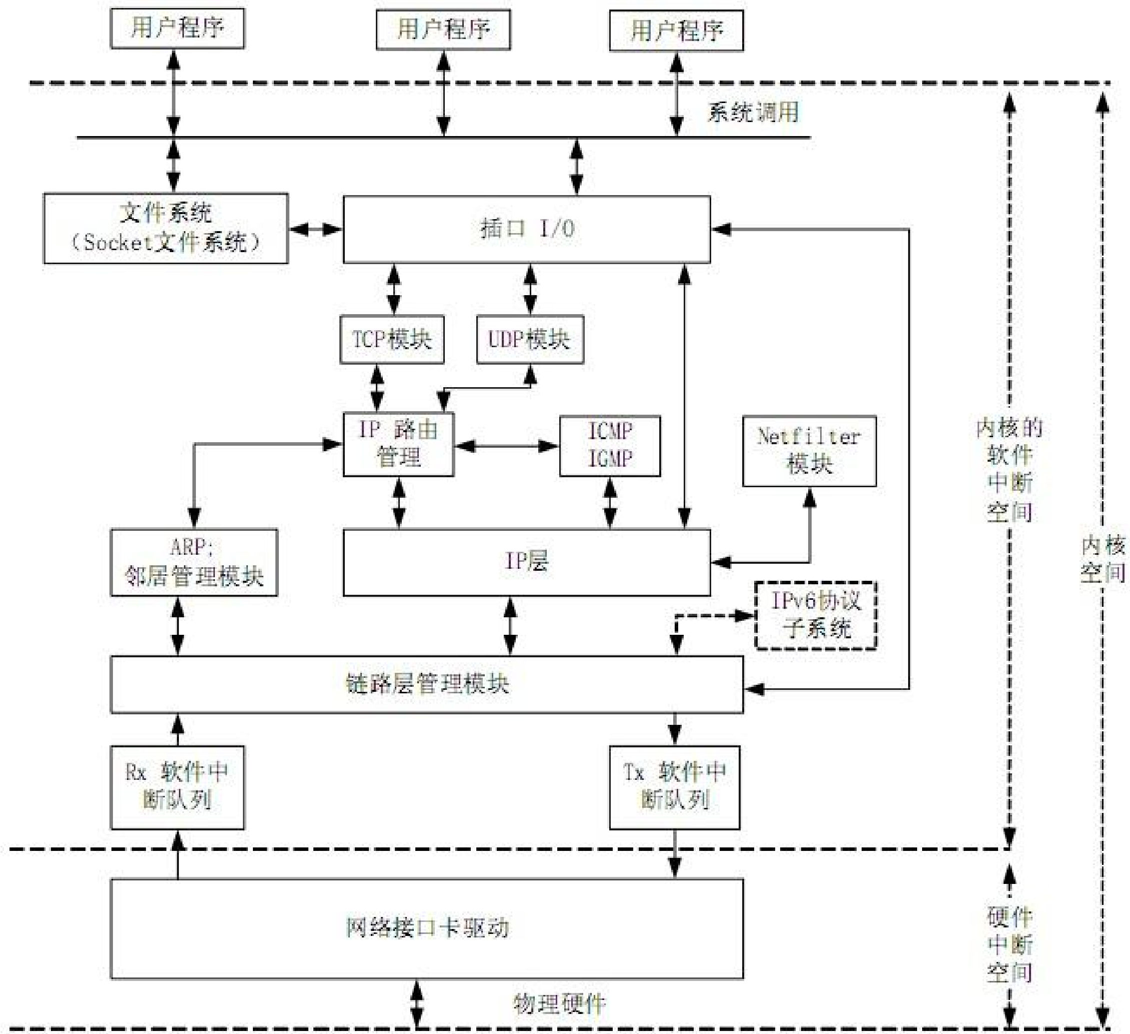


图1 Linux 内核的 TCP/IP 的体系结构图

Allocation Structures

- Allocation structures describe the mapping from software structures to the system's environments
- For example
  - Modules are assigned to **teams** to develop, and assigned to places in **a file structure** for implementation, integration, and testing.
  - Components are deployed onto **hardware** in order to execute.

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.
- 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
  - the responsiveness of the system to user requests,
  - many others.

Architecture is an Abstraction

- An architecture specifically omits certain information about elements that is not useful for reasoning about the system.
- 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, and so forth.
- This abstraction is essential to taming the complexity of an architecture.

Every System has a Software Architecture

- But the architecture may not be known to anyone.
  - Perhaps all of the people who designed the system are long gone
  - Perhaps the documentation has vanished (or was never produced)
  - Perhaps the source code has been lost (or was never delivered)

Architecture Includes Behavior

- The behavior of each element is part of the architecture insofar as that behavior can be used to reason about the system.
- This behavior embodies how elements interact with each other, which is clearly part of the definition of architecture.

Module Structures

- **Module structures** embody decisions as to how the system is to be structured as a set of code or data units
- 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.

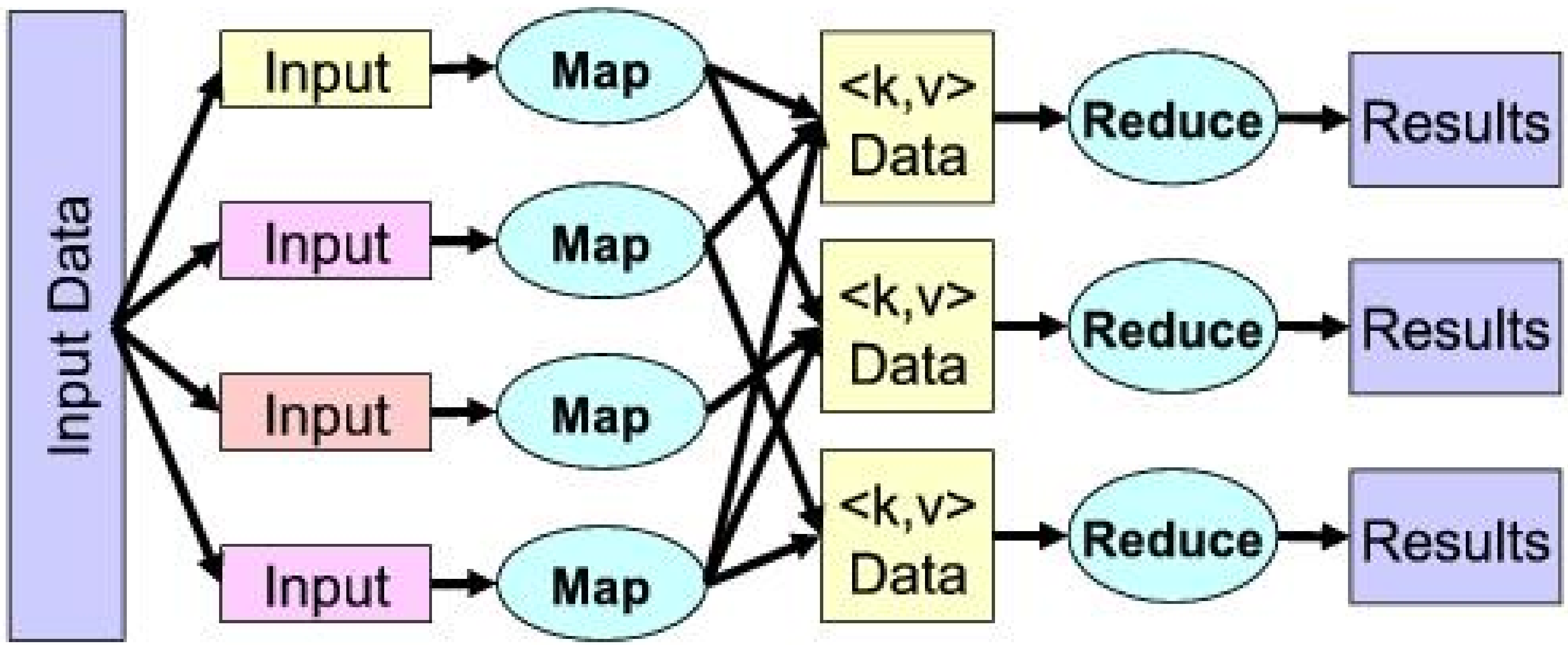
Component-and-connector Structures

- 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 comopnents such as services, peers, clients, servers, or many other types of runtime element)
- Connectors are the communication ways among components, such as call-return, process synchronization operators, pipes, or others.

Component-and-connector Structures

- *Component-and-connector structures* 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?

MapReduce



Input Data
Input
Input
Input
Input
Map
Map
Map
Map
<k,v> Data
<k,v> Data
<k,v> Data
Reduce
Reduce
Reduce
Results
Results
Results

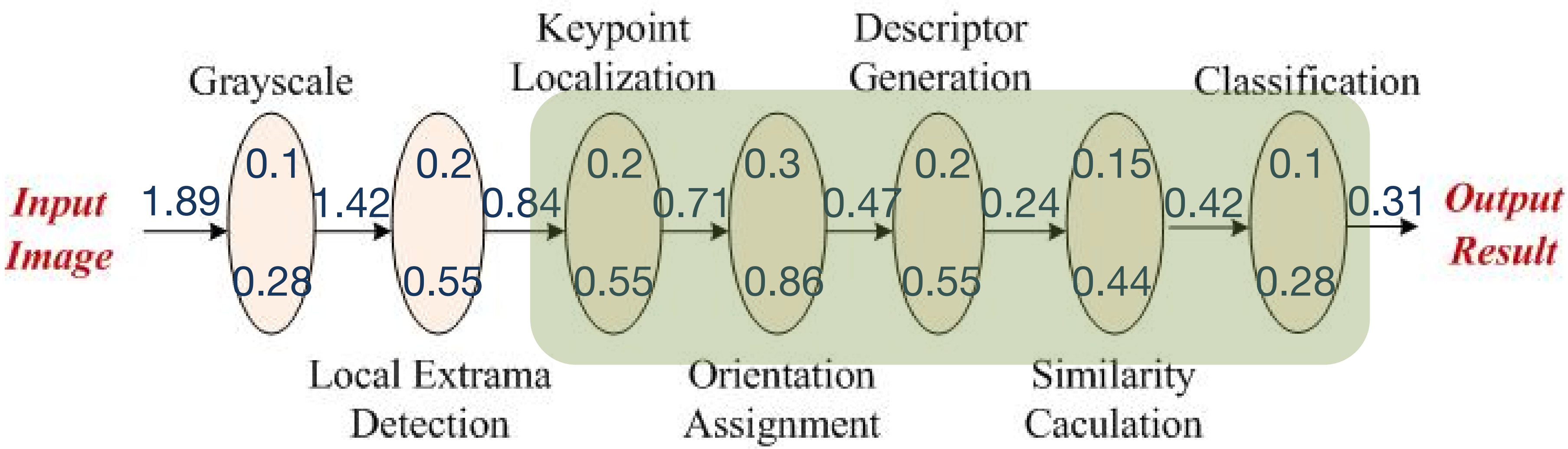
Allocation structures

- ***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 structure*** 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.
  - 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.
  - The **deployment structure** is strongly tied to the achievement of performance, availability, and security goals.

Computation Partitioning
a simple example



Optimal Partitioning 0.28 + 0.55 + 0.84 + $\underline{0.2 + 0.3 + 0.2 + 0.15 + 0.1}$ + 0.31 = 2.93

Local Execution: 0.28 + 0.55 + 0.55 + 0.86 + 0.55 + 0.44 + 0.28 = 3.51

Remote Execution: 1.89 + $\underline{0.1 + 0.2 + 0.2 + 0.3 + 0.2 + 0.15 + 0.1}$ + 0.31 = 3.45

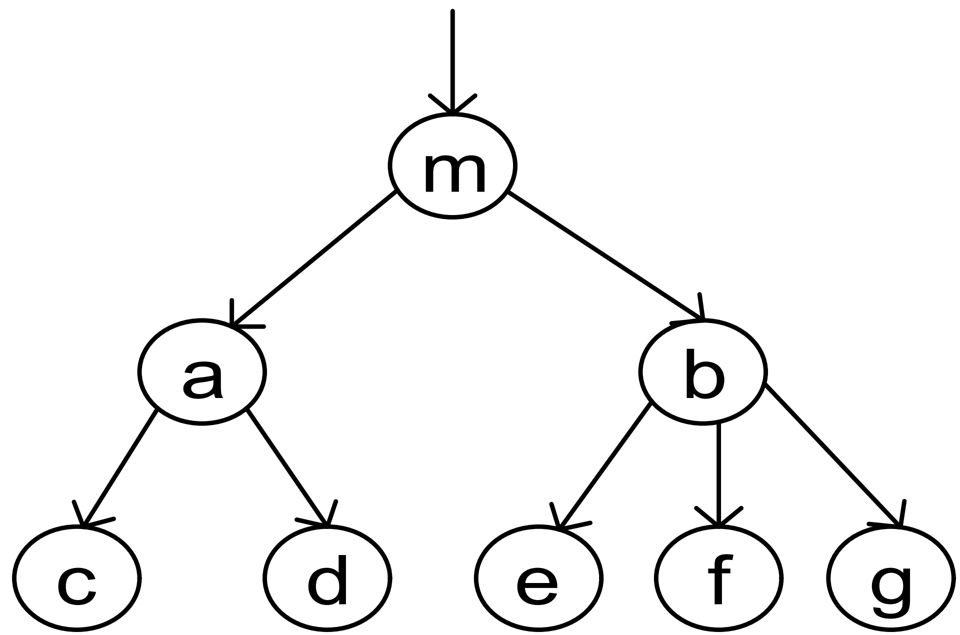
Some Useful Module Structures

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.

Decomposition structure: an example

```
void m()
{
    a() {
        c();
        d();
    };
    b() {
        e();
        f();
        g();
    };
}
```

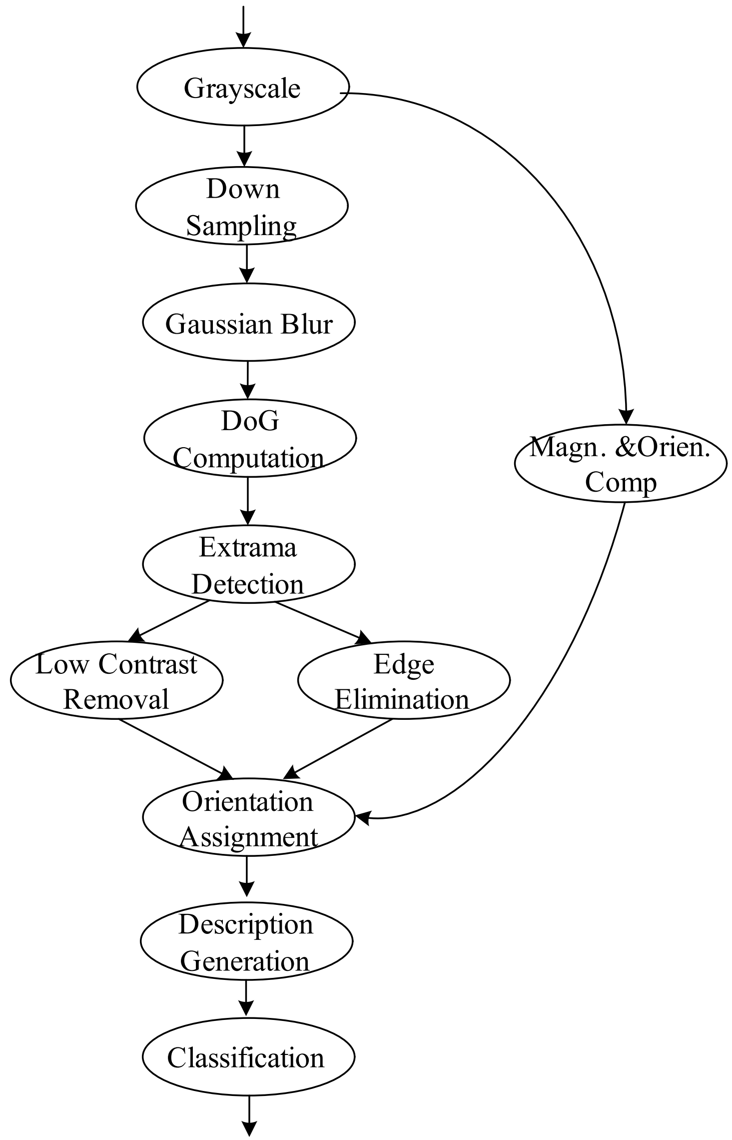


Some Useful Module Structures

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 of the second.

User structure: an example



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 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*.
- ***Inheritance*** is a mechanism for code reuse and to allow independent extensions of the original software
- The class structure allows one to reason about reuse and the incremental addition of functionality.

Some Useful Module Structures

Data model structure

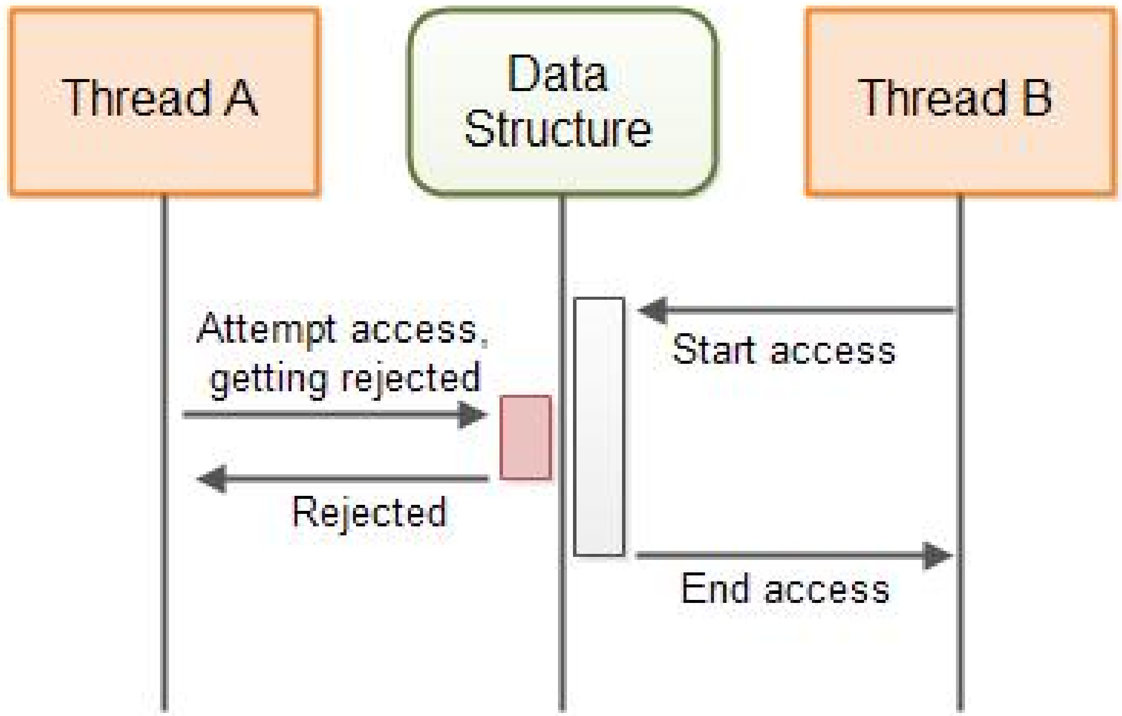
- The data model describes the static information structure in terms of data entities and their relationships
  - For example, in a banking system, entities will typically include Account, Customer, and Loan.
  - Account has several attributes, such as account number, type (savings or checking), status,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

Some Useful C&C Structures

- 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.

Some Useful C&C Structures

- 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.

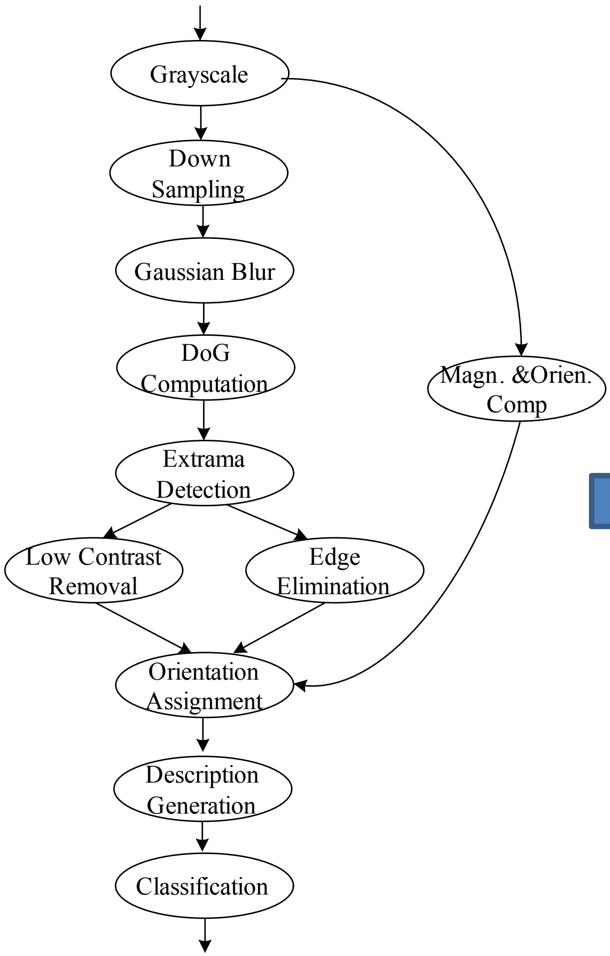

Thread A
Data Structure
Thread B
Attempt access, getting rejected
Rejected
Start access
End access

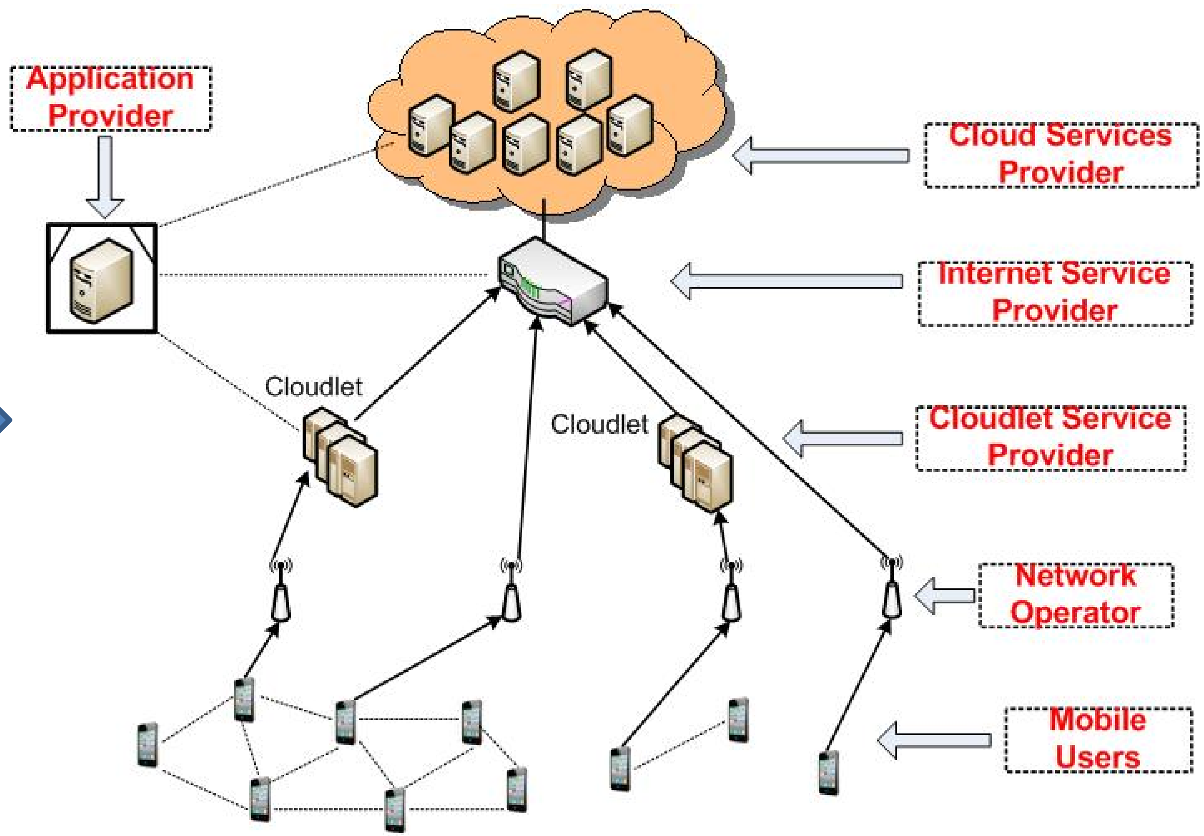
Some Useful Allocation Structures

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.

Task allocation in mobile cloud



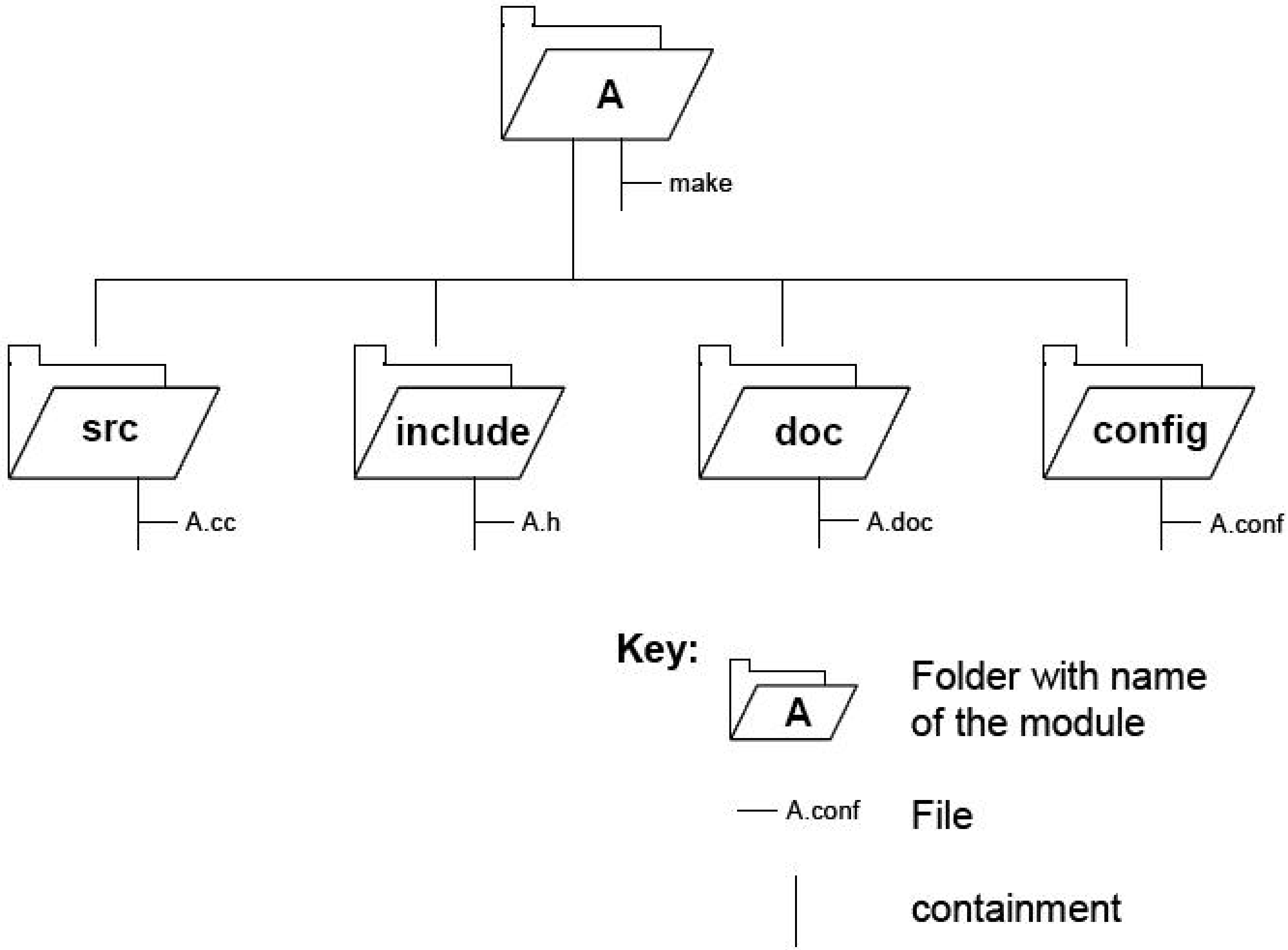


Some Useful Allocation Structures

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.

Implementation structure



Some Useful Allocation Structures

Work assignment structure

- This structure assigns responsibility for implementing and integrating the modules to the teams who will carry it out.

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

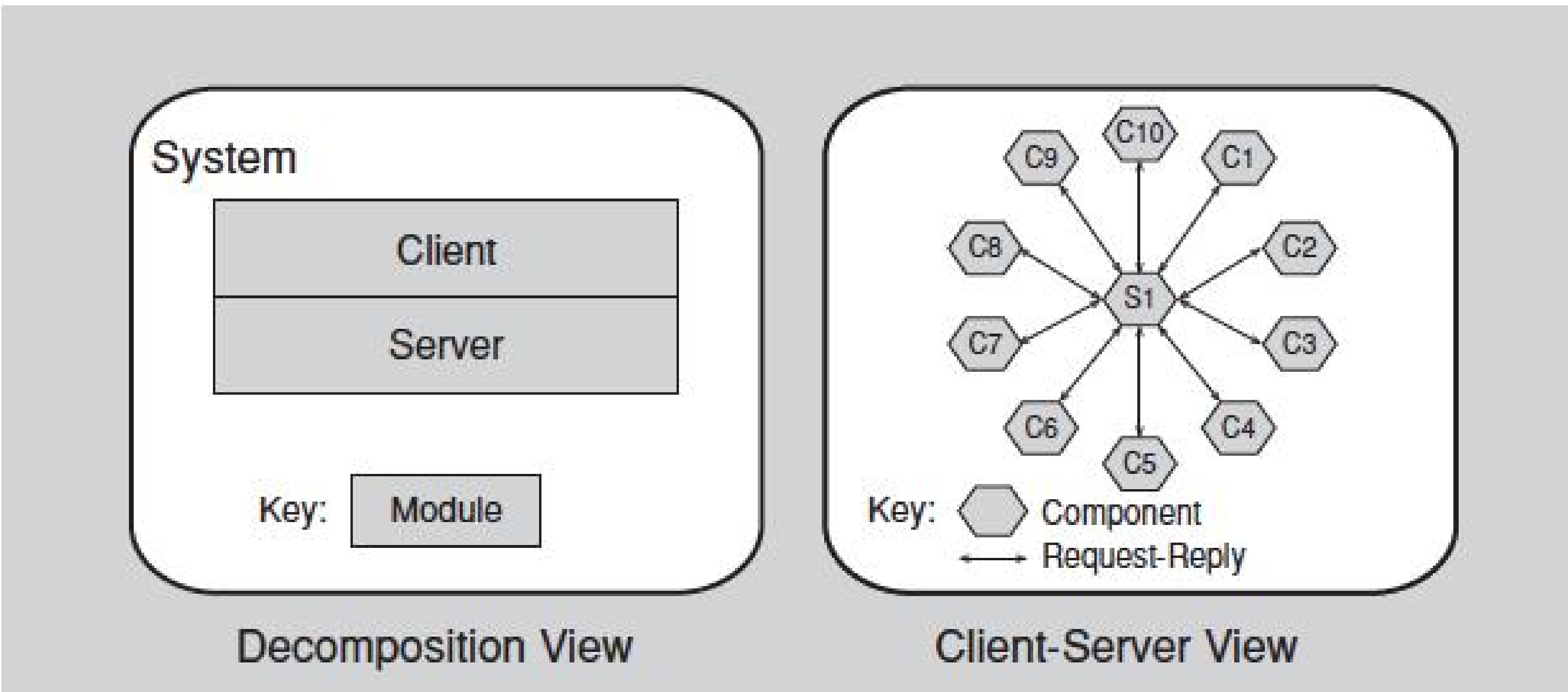


FIGURE 1.2 Two views of a client-server system

Architectural Patterns

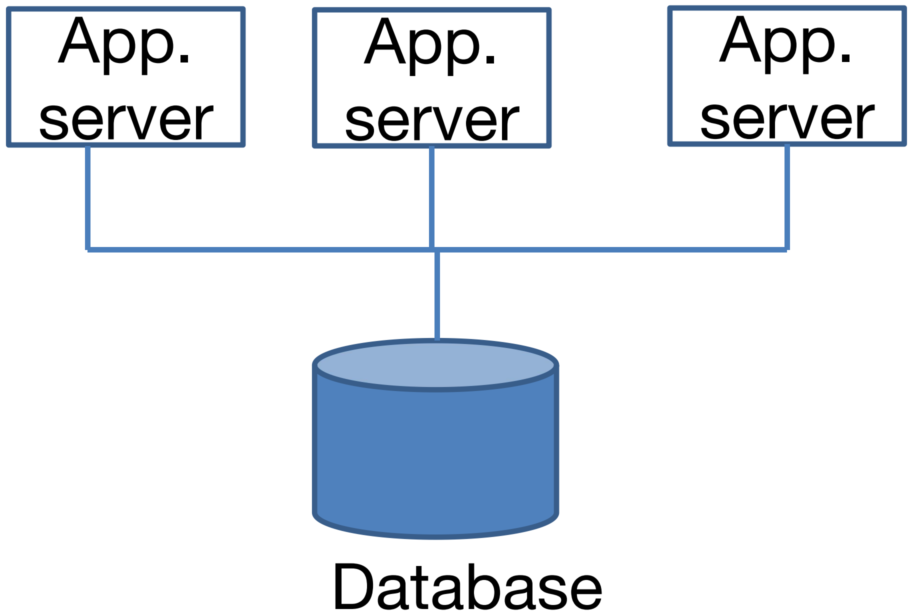
- An architectural pattern presents the element types and their forms of interaction used in solving a particular problem.
- A common *module type* pattern is the Layered pattern.
  - When the uses relation among software elements is strictly unidirectional, a system of layers emerges.
  - A layer is a coherent set of related functionality.

Architectural Patterns

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.

Shared data pattern



Architectural Patterns

Common *component-and-connector* type patterns:

- 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.
- Peer-to-peer pattern
  - E.g. Bittorrent, eMule

Architectural Patterns

Common allocation patterns:

- Multi-tier pattern
  - This pattern specializes the generic deployment (software-to-hardware allocation) structure.
  - Describes how to distribute and allocate the components of a system in distinct subsets of hardware and software, connected by some communication medium.

Architectural Patterns

Common allocation patterns:

- *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.

What Makes a “Good” Architecture?

- There is no such thing as an inherently good or bad architecture.
- Architectures can be evaluated but only in the context of specific stated goals.
- There are, however, good rules of thumb.

Process "Rules of Thumb"

- The architecture should be the product of a single architect or a small group of architects with an identified technical leader.
- The architect (or architecture team) should base the architecture on a prioritized list of well-specified quality attribute requirements.
- The architecture should be documented using views.
- The architecture should be evaluated for its ability to deliver the system's important quality attributes.
- The architecture should lend itself to incremental implementation.

Structural "Rules of Thumb"

- The architecture should feature well-defined modules
- The architecture should never depend on a particular version of a commercial product or tool
- Modules that produce data should be separate from modules that consume data.
  - This tends to increase modifiability

Structural "Rules of Thumb"

- Don't expect a one-to-one correspondence between modules and components.
- Every process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.
- The architecture should feature a small number of ways for components to interact.
  - The system should do the same things in the same way throughout.

Summary

- ***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.
- ***A structure*** is a set of elements and the relations among them.
- ***A view*** is a representation of a coherent set of architectural elements. A view is a representation of one or more structures.

Some Useful Structures

- Module structures
  - Decomposition structure
  - User structure -> layer pattern
  - Class structure
  - Data model
- Component-and-connector structures
  - Service structure
  - Concurrency structure
- Allocation structures
  - Deployment structure
  - Implementation structure
  - Work assignment structure

Architectural patterns

- Module type pattern
  - Layered pattern
- Component-and-connector type pattern
  - Shared data pattern
  - Client and server pattern
  - Peer to peer pattern
- Allocation type pattern
  - Multi-tier pattern
  - Competence center pattern
  - Platform pattern

Class Review

1. Which one of the following statement is NOT true about software structures?

- ✓ A. Software systems are composed of many structures, and no single structure holds claim to being the architecture
- ✓ B. The module structure shows how each of the software elements behaves at run time
- ✓ C. The component & connector structure shows how the software elements are interacted at run time
- ✓ D. The deployment structure shows information about at which physical hardware the software elements are executed

2. Which one of the following structure pertains to a component & connector structure?

- ✓ A. Decomposition structure
- ✓ B. Class structure
- ✓ C. Layer structure
- ✓ D. Concurrency structure

3. Which of the following should be included in the consideration of a Module Structure?

- ✓ A. What are the major executing components and how do they interact at runtime
- ✓ B. what part of the system can run in parallel
- ✓ C. What is the primary functional responsibility assigned to each software elements
- ✓ D. How does data process through the system

4. Which one of the following structures pertains to the Allocation Structure?

- ✓ A. Service structure
- ✓ B. Concurrency structure
- ✓ C. Layer structure
- ✓ D. Deployment structure