# Software Architecture

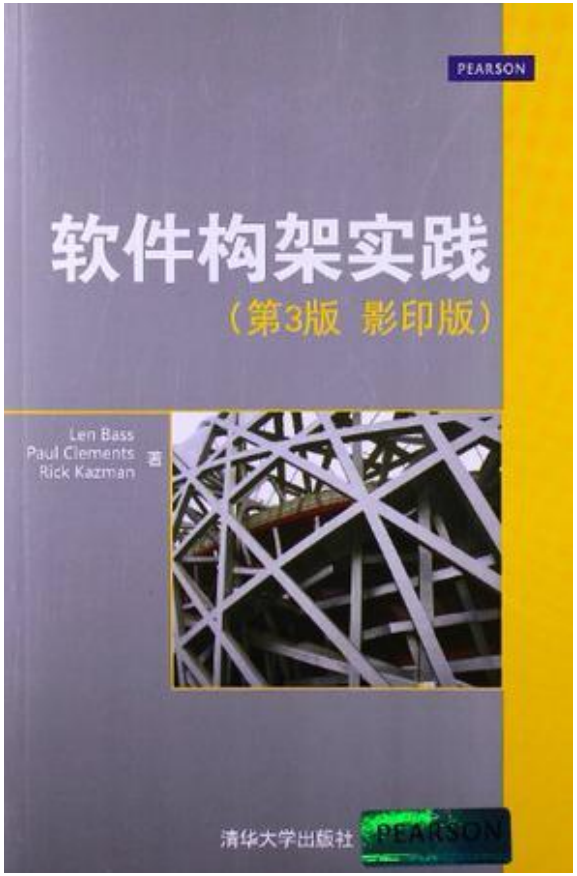
Lecturer： Dr. Lei Yang
sely@scut.edu.cn
South China University of Technology

# 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:
  - 软件构架编档，Paul Clements，Felix Bachmann 等著，朱崇高 译，清华大学出版社(2004)
  - 软件体系结构——一门初露端倪学科的展望,M. Shaw and D. Garlan, Prentice Hall, 1996 清华大学出版社(1998), 科学出版社(2003)
  - 软件体系结构，张友生等，清华大学出版社，2006



# 考试与成绩

- 期末考试 60%
- 平时成绩 40%
  - 大作业
  - 课堂考勤
  - 课后习题

# Major Contents

- Introduction of software architecture
  - what, why, contexts of software architecture
- Quality attributes
  - Availability, modifiability, performance, security, testability, and usability
- Architecture in the life cycle
  - Requirement, design, implementation, test, and evaluation
- Architecture in the Cloud

# 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*.
- In our use, 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.


应用程序
Home
联系人
电话
浏览器
...
应用程序框架
Activity管理器
Window管理器
内容提供者
View系统
包管理器
电话管理器
资源管理器
定位管理器
消息管理器
核心库
显示系统管理
媒体框架
SQLite数据库
OpenGL | ES
FreeType
WebKit
SGL
SSL
C库
Android 运行时
运行时核心库
Dalvik虚拟机
Linux 内核
显示驱动
摄像头驱动
闪存驱动
绑定(进程间通讯)驱动
键盘驱动
WiFi 驱动
音频驱动
电源管理

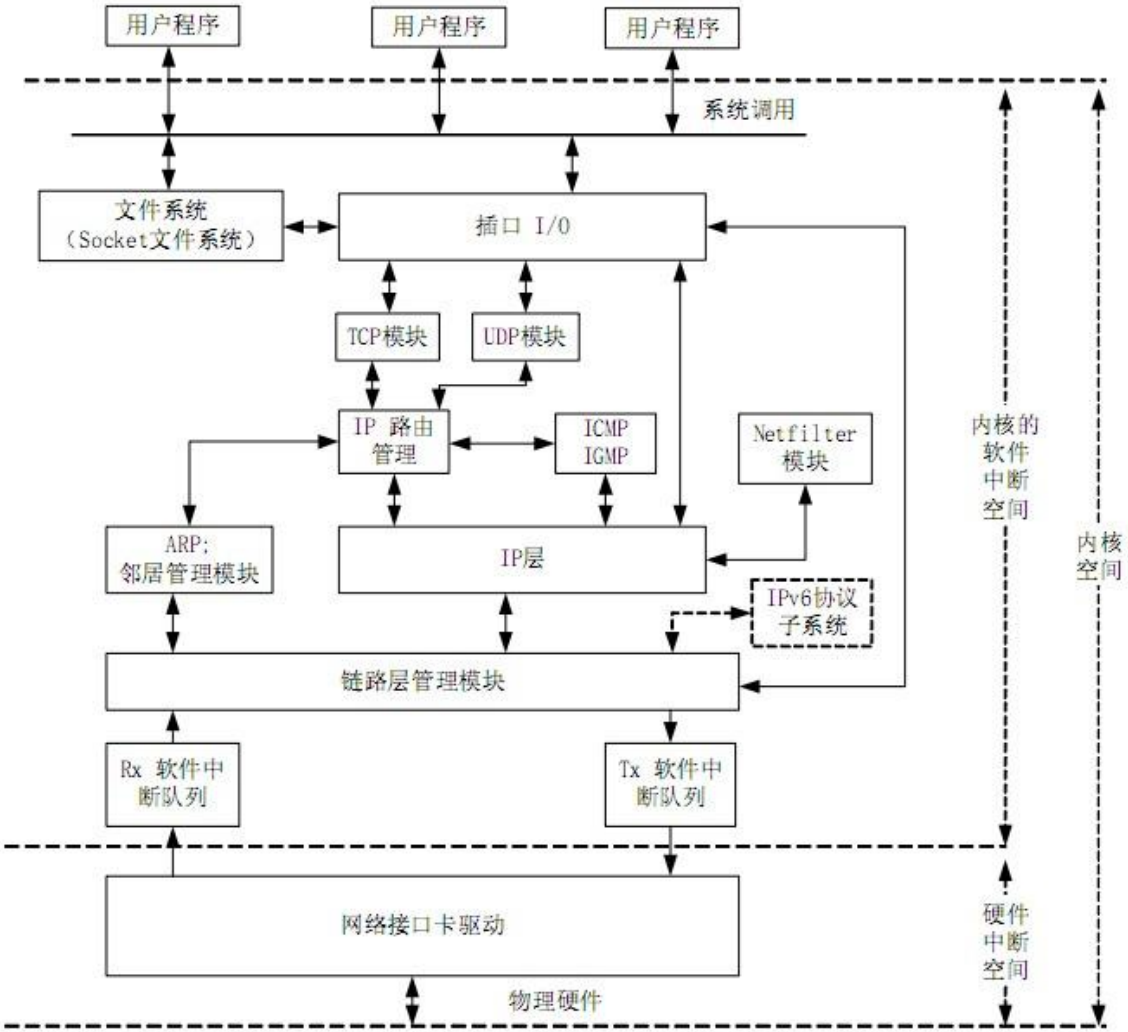



图1 Linux 内核的 TCP/IP 的体系结构图

# 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)
- An architecture can exist independently of its description or specification

# 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.
- This does not mean that the exact behavior and performance of every element must be documented in all circumstances.

# Structures and Views

- A *view* is a representation of a coherent set of architectural elements, as written by and read by system stakeholders.
- A *structure* is the set of elements itself, as they exist in software or hardware.
- In short, a view is a representation of a structure.
  - For example, a module *structure* is the set of the system's modules and their organization.
  - A module *view* is the representation of that structure, documented according to a template in a chosen notation, and used by some system stakeholders.
- Architects design structures. They document views of those structures.

# 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.

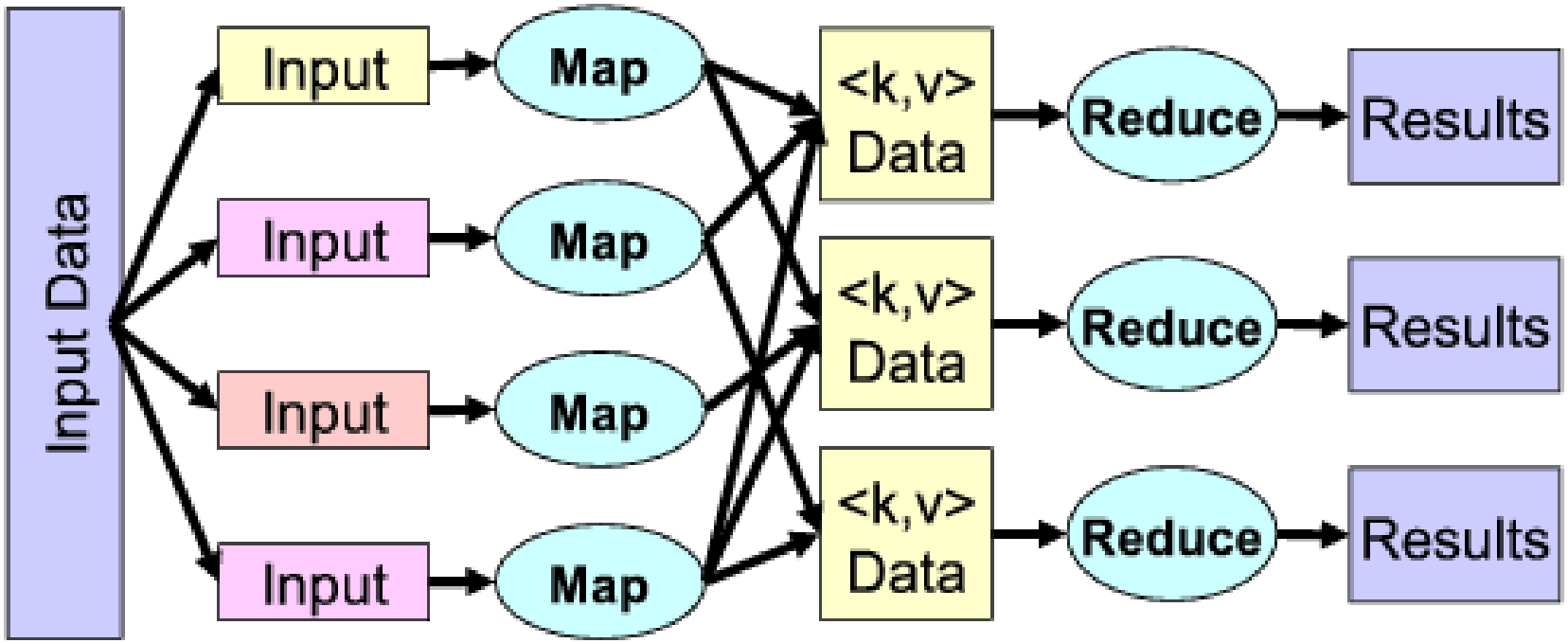
# Component-and-connector Structures

- 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 vehicles among components, such as call-return, process synchronization operators, pipes, or others.

# Component-and-connector Structures

- *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?

# 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

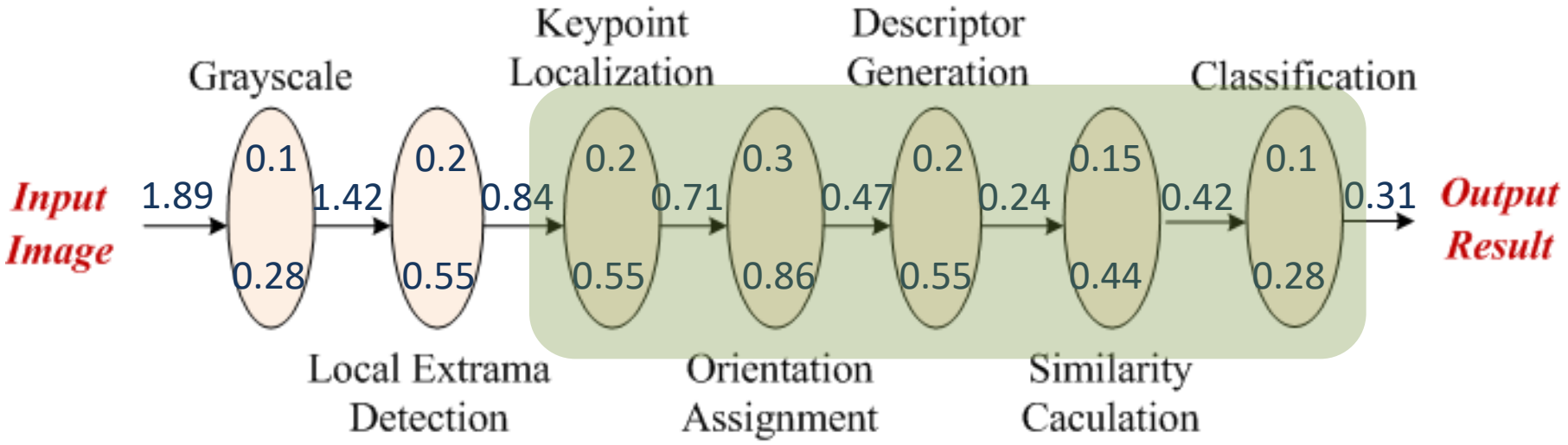
# Allocation structures

- ***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 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.
  - 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 + 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 + 0.1 + 0.2 + 0.2 + 0.3 + 0.2 + 0.15 + 0.1 + 0.31 = 3.45

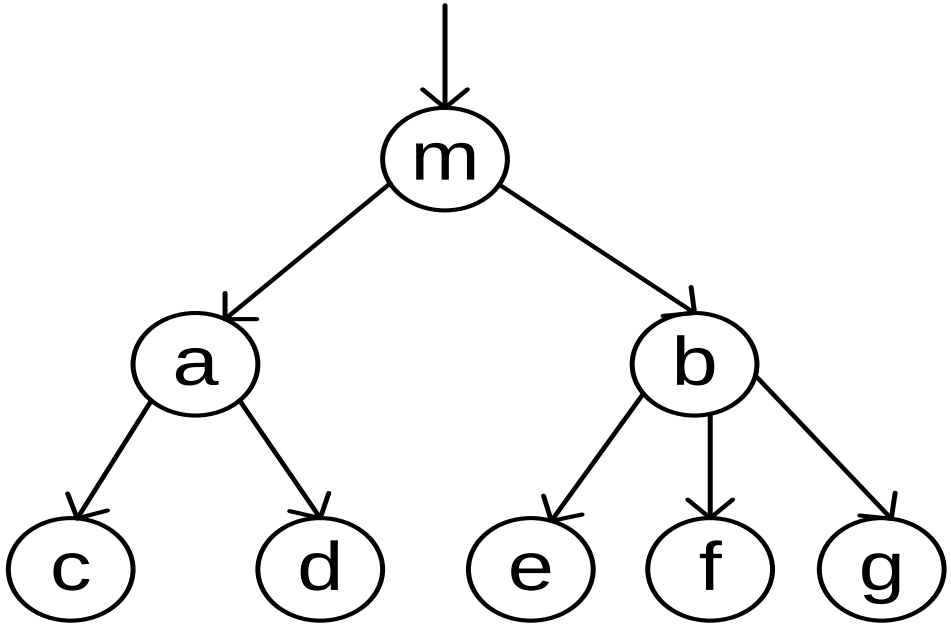
# Some Useful Module Structures

**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();
    };
}
```

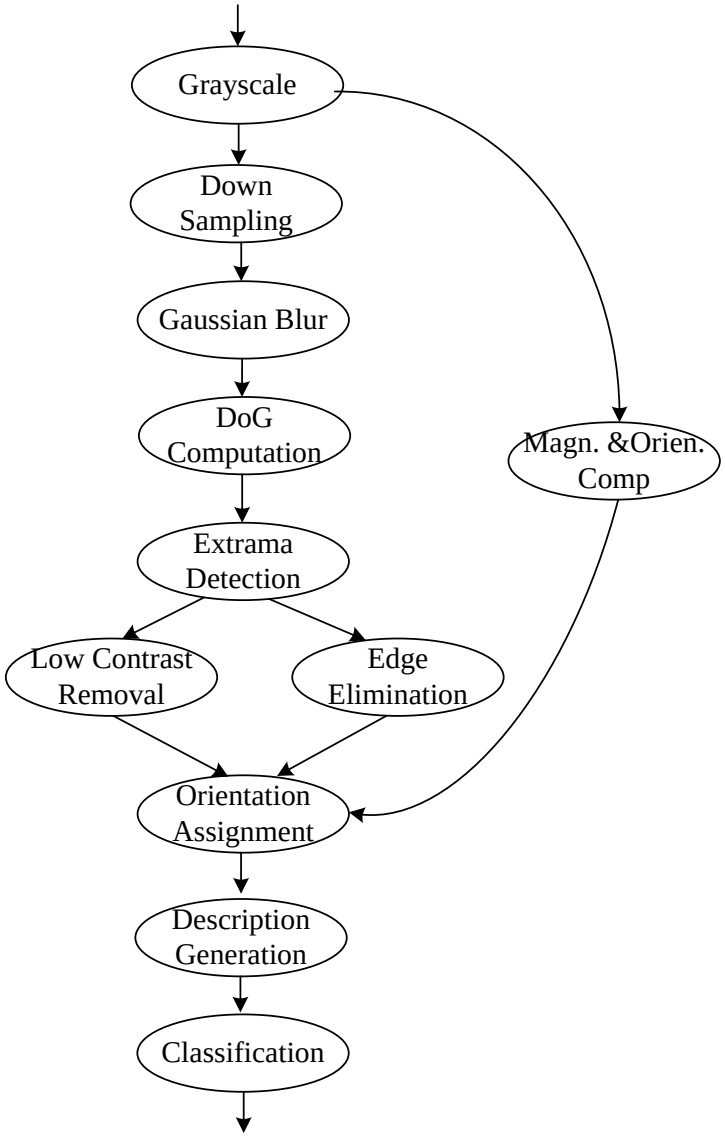


# Some Useful Module Structures

**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 (as opposed to a stub) of the second.
- The ability to easily create a subset of a system allows for incremental development.

# 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 is imbues a system with portability, the ability to change the underlying computing platform.


应用程序
Home
联系人
电话
浏览器
...
应用程序框架
Activity管理器
Window管理器
内容提供者
View系统
包管理器
电话管理器
资源管理器
定位管理器
消息管理器
核心库
显示系统管理
媒体框架
SQLite数据库
OpenGL | ES
FreeType
WebKit
SGL
SSL
C库
Android 运行时
运行时核心库
Dalvik虚拟机
Linux 内核
显示驱动
摄像头驱动
闪存驱动
绑定(进程间通讯)驱动
键盘驱动
WiFi 驱动
音频驱动
电源管理

# 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

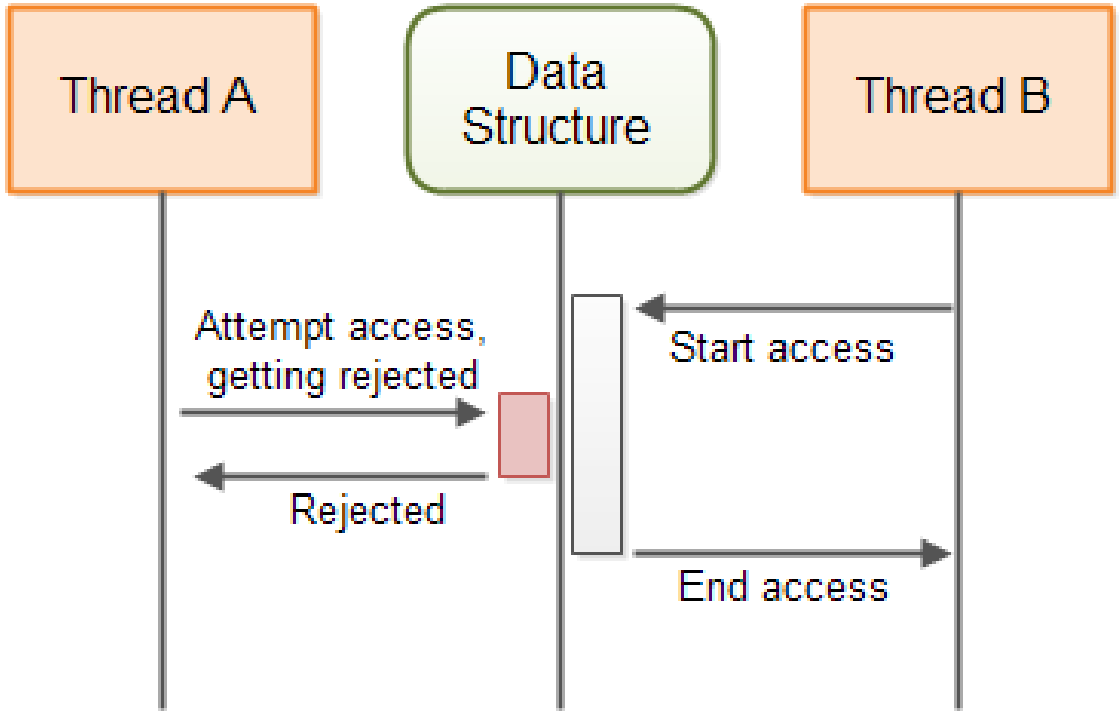
- The data model describes the static information structure in terms of data entities and their relationships
  - For example, in a banking system, entities will typically include Account, Customer, and Loan.
  - Account has several attributes, such as account number, type (savings or checking), status,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

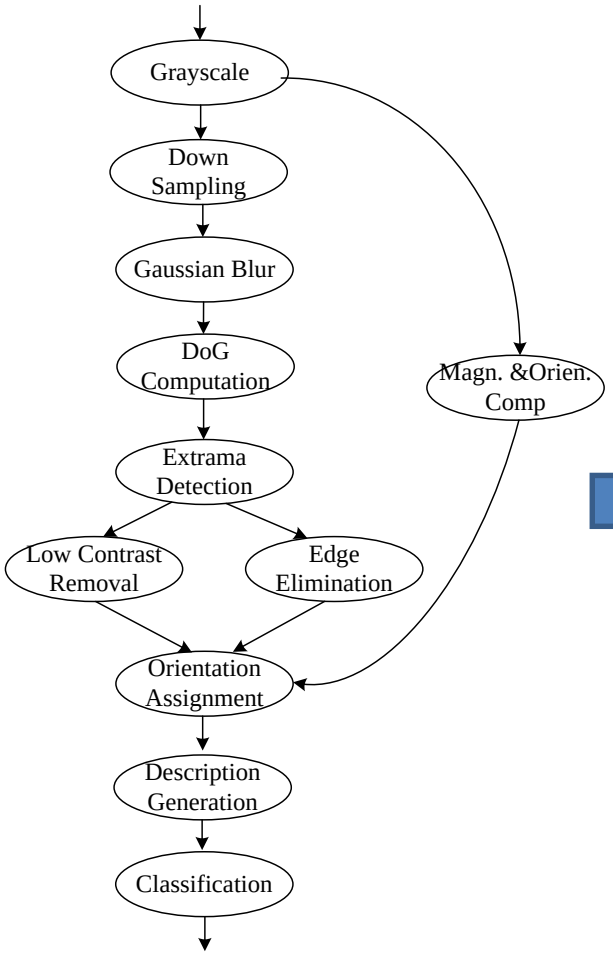

Thread A
Data Structure
Thread B
Attempt access, getting rejected
Start access
Rejected
End access

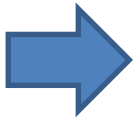
# Some Useful Allocation Structures

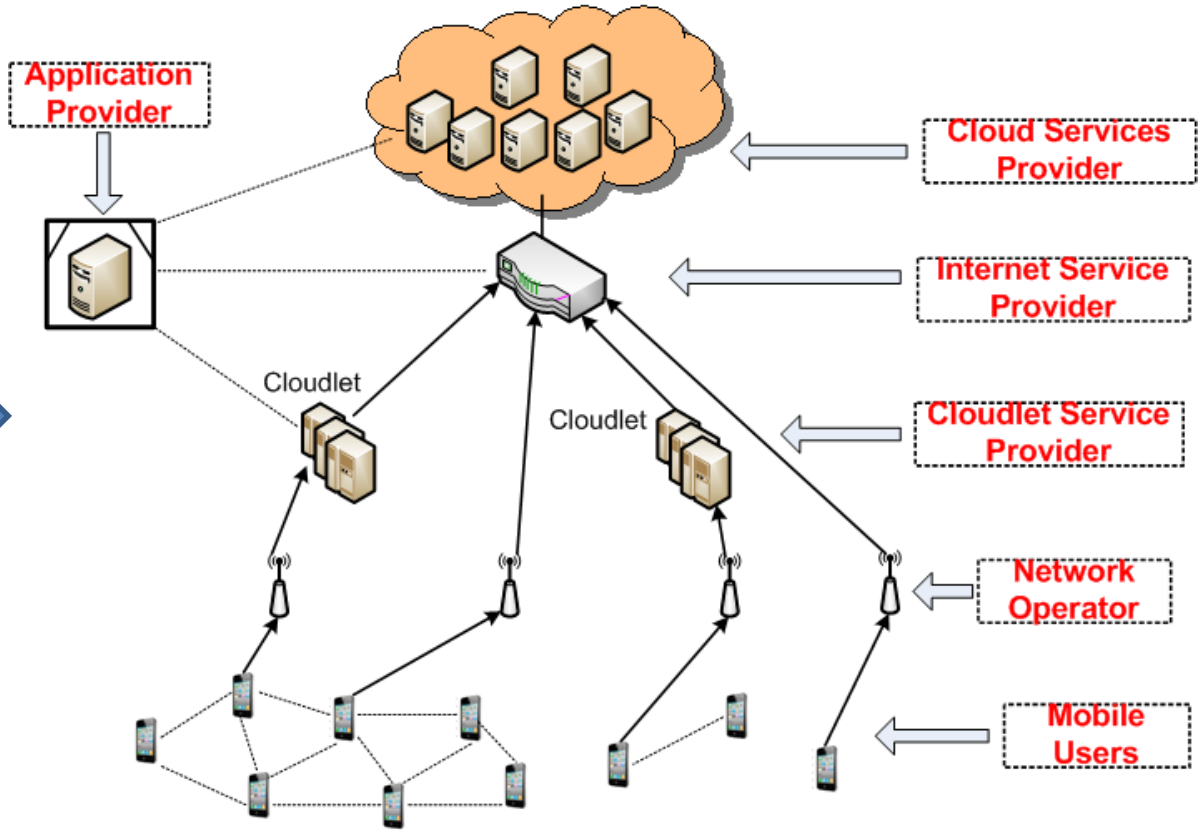
**Deployment structure**

- 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



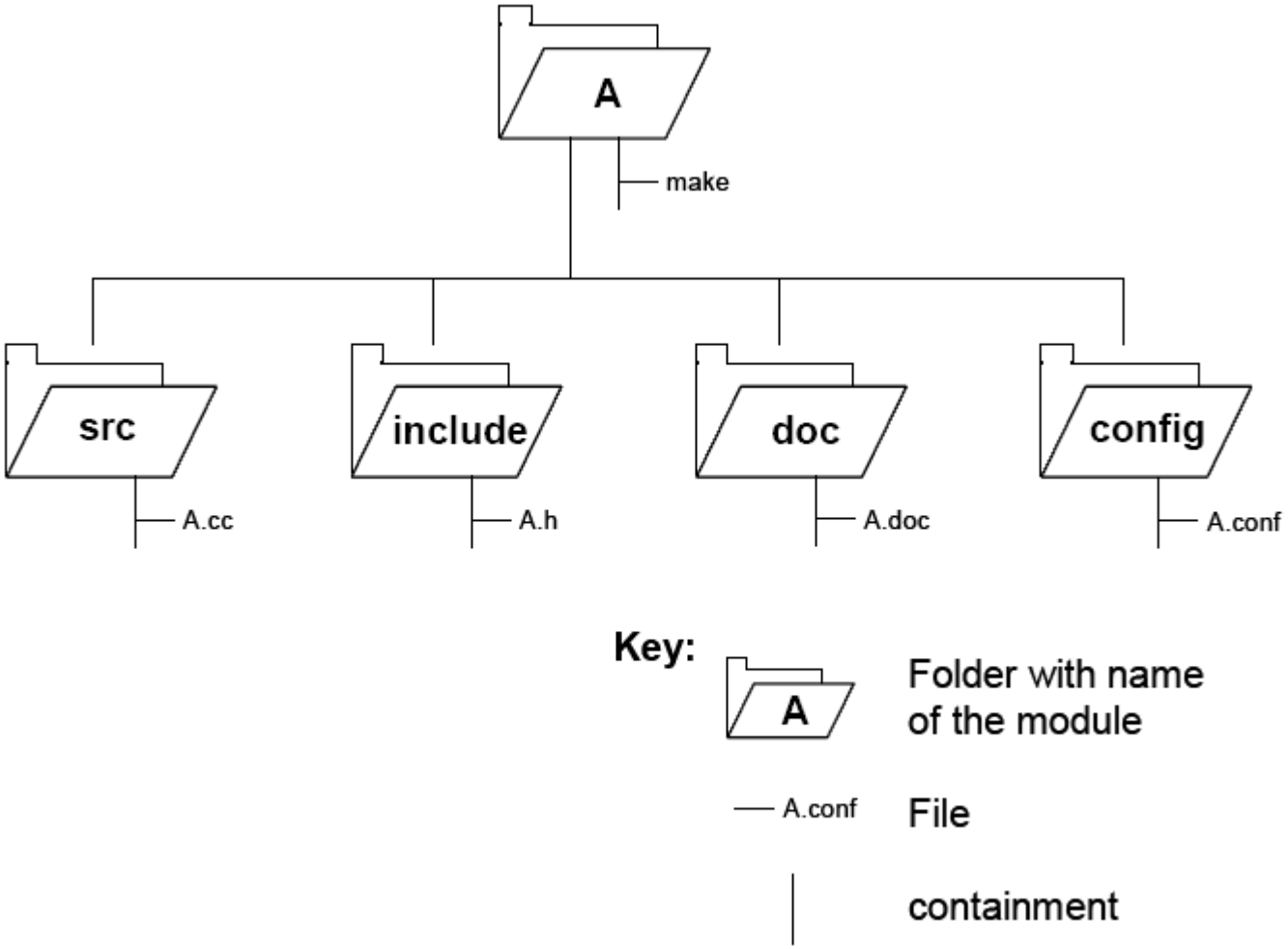


# Some Useful Allocation Structures

**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

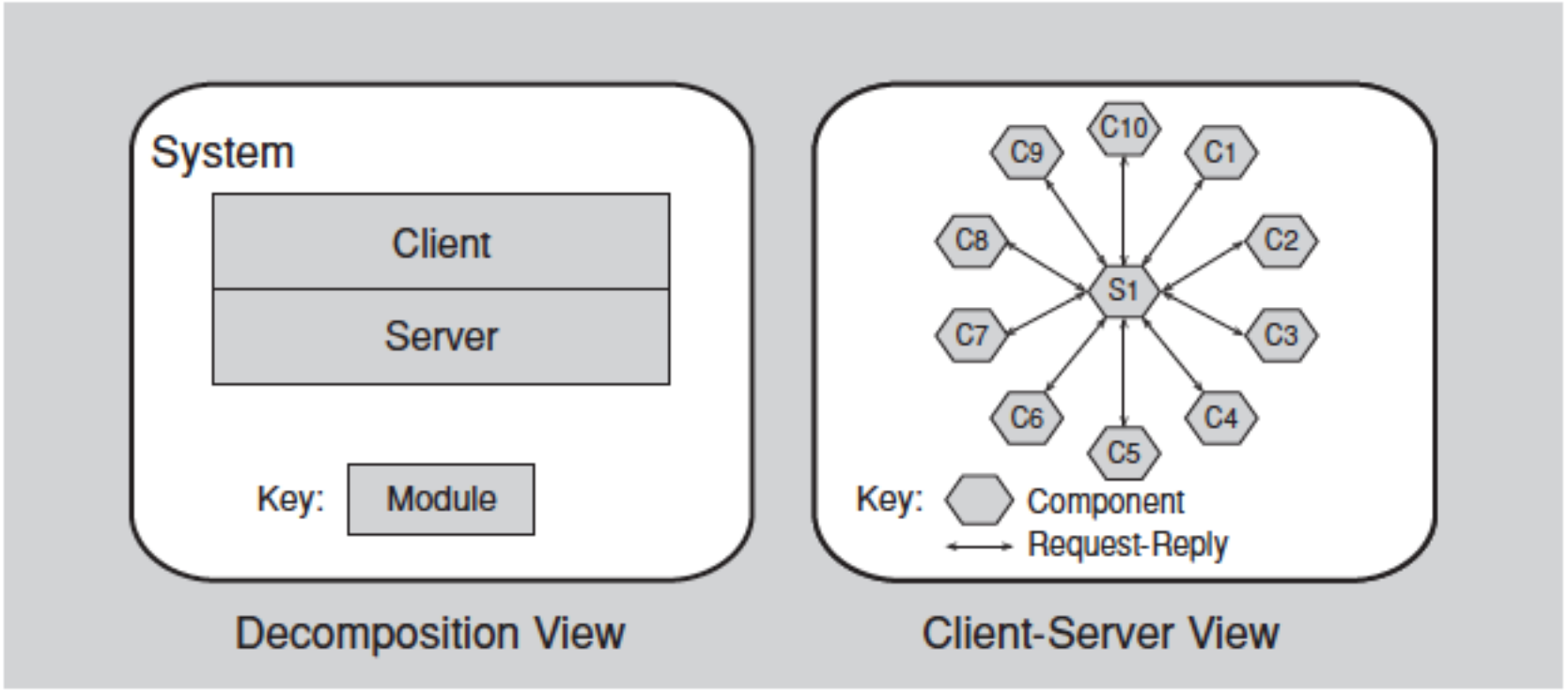



**FIGURE 1.2** Two views of a client-server system

# Architectural Patterns

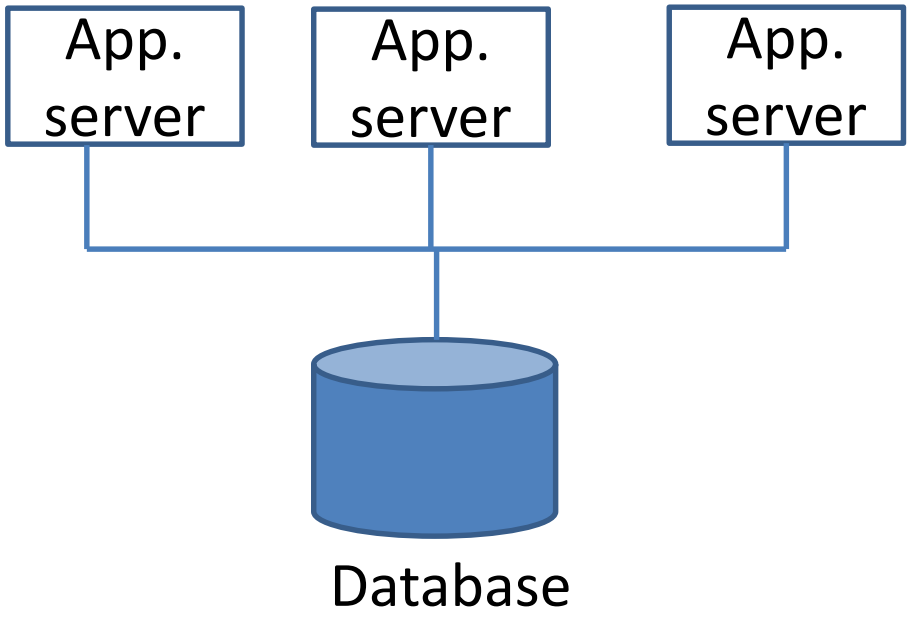
- An architectural pattern presents the element types and their forms of interaction used in solving a particular problem.
- A common *module type* pattern is the Layered pattern.
  - When the uses relation among software elements is strictly unidirectional, a system of layers emerges.
  - A layer is 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.
- Architectures are either more or less fit for some purpose
- 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