Software Architecture

Part I

Context

Course

- Unit 1: Agile planning, design, project management
- **Unit 2: Cloud software, architectural design**
  - Going into more depth on topics we've already touched on

Project

- Sprint 1—Designs & technologies
- Sprint 2—Development: Back-end data service
  - **Dev: data manager**
  - Dev: data service
  - Test, deploy: Data service

Outline

- Software architecture
- System quality attributes
- Design issues and trade-offs
- System decomposition
- Service-oriented architectures (SOA)

Software architecture

Fundamental organization of a software system embodied in…

- Its **components**,
- Their **relationships** to each other and to the environment, and
- The **principles** guiding its design and evolution.

(IEEE)

Component

- Coherent set of functionality/features
  - Main focus: interface to component functionality
  - Can be implemented later
    - …or changed later
    - …even as systems that depend on it are implemented
- May also be thought of in terms of data and the services surrounding that data

Components in context

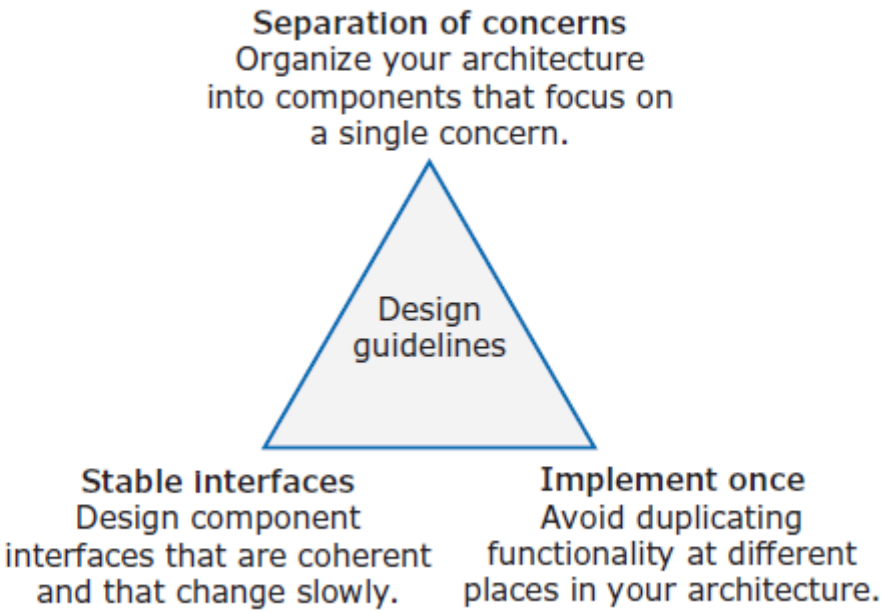
- Component-based software engineering
  - Emphasizes **separation of concerns** in larger software system
  - Enables **reuse**: can compose loosely coupled independent components into systems
- Idea of components key for service-orientation
  - Web services
  - Service-oriented architectures (SOA)

Component relationships

- Depict connections between components
  - Could be structural, behavioral, or grouping
- Demonstrate how components are interrelated at *time of execution* (run-time)
- Important examples:
  - Composition (“has-a”)
  - Uses (makes calls to)
  - Generalization/realization (“is-a”)

Principles of component design

Figure 4.8 Architectural design guidelines



Separation of concerns

- Components should be separated based on the **kinds of work** performed
- Related: **single responsibility**
  - Components should have one responsibility and one reason to change
  - Only reason to change is if responsibility updated
- Key benefit: easier to work with in isolation

Implement once

- Avoid duplicating functionality at different places in your system
- **DRY**—Don’t repeat yourself
- Key Benefit: one place to fix problem or add nuances to behavior (vs. all the copies)

Stable interfaces

- Design coherent interfaces (methods or API's) for components that *change slowly*
- Related: **encapsulation**
- Related: **dependency inversion**
  - A component should depend on an abstraction (interface), not on implementation details
- Key benefit: changes to implementation do not require changes to rest of system

Plus: Database concerns

- Persistence ignorance (PI)
  - Refers to data that need to be persisted, but whose code is unaffected by the choice of persistence technology. (i.e., database)
- Bounded contexts
  - Each component represents a context that is separated from other contexts (hence, bounded), and can evolve independently
  - Each should be free to choose own names for concepts within it, have exclusive access to its own persistence store (database)

Key questions for arch. design

- Organization
  - How should system components be organized?
    - So that each provides subset of overall functionality
    - Delivers required security, reliability, performance
- Distribution
  - How are components distributed?
  - How do they communicate?
- Technologies—which ones? Lead to reuse?

Outline

- Software architecture
- System quality attributes
- Design issues and trade-offs
- System decomposition
- Service-oriented architectures (SOA)

Prototype vs. product dev.

- First: develop product **prototype**
  - Help you better understand end product
  - Speed is paramount
  - Not important: security, usability, maintenance…
- Later: develop actual **product**
  - "Non-functional" attributes become critically important (Table 4.2)
  - Development takes much longer to ensure these

Important system quality attrib's

Table 4.2 Non-functional system quality attributes

Attribute	Key issue
Responsiveness	Does the system return results to users in a reasonable time?
Reliability	Do the system features behave as expected by both developers and users?
Availability	Can the system deliver its services when requested by users?
Security	Does the system protect itself and users' data from unauthorized attacks and intrusions?
Usability	Can system users access the features that they need and use them quickly and without errors?
Maintainability	Can the system be readily updated and new features added without undue costs?
Resilience	Can the system continue to deliver user services in the event of partial failure or external attack?

Common SQA subset: Agility

- *Agility* (n.)
  1. ability to move quickly, easily
  2. ability to think and understand quickly
- Agility in software architecture aggregates 7 architecturally sensitive attributes
  - debuggability
  - extensibility
  - portability
  - scalability
  - securability
  - testability
  - understandability

How do Agile planning and development principles relate to these attributes?

Other common SQA subsets

- **Dependability**: availability, reliability, safety, integrity and maintainability
- **Integrity**: security and survivability
- **Security**: confidentiality, integrity and availability

  Security and dependability are often treated together.

Outline

- Software architecture
- System quality attributes
- Design issues and trade-offs
- System decomposition
- Service-oriented architectures (SOA)

Architectural design process

- **Understand** issues that affect the architecture of your product
- Begin **making design** decisions
- **Create descriptions** of architecture showing critical components and relationships

Design decisions, trade-offs

Architectural design involves

- Deciding on essential compromises that allow you to create system that
  - Is "**good enough**"
  - Can be delivered on-time, on-budget
- Making **trade-offs** in architecture choices:
  - maintainability vs. performance
  - security vs. usability
  - availability vs. time to market & cost

Some architectural design issues

- Product lifetime—long lifetime requires architecture that can evolve (adaptability)
- Software reuse—Commercial or open-source software can save time, …but may constrain design (vs. home-grown solution)
- Number of users—Scalability to handle wide swings in # of users
- Software compatibility—Working with other systems may constrain your design

Outline

- Software architecture
- System quality attributes
- Design issues and trade-offs
- System decomposition
- Service-oriented architectures (SOA)

System decomposition

- At architectural level: concern should be on **large-scale** architectural components
- **Decomposition** involves analyzing larger-scale components and representing them as a set of finer-grain components

You've started this for your project, but let's look at another example

Example

Identify key noun-phrases
→ components

- Document retrieval system

"This system was designed for use in libraries and gave users access to documents that were stored in a number of private databases, such as legal and patent databases. Payment was required for access to these documents. The system had to manage the rights to these documents and collect and account for access payments."

Identify key verb-phrases
→ relationships

Breaking it down

Context, Personas/users

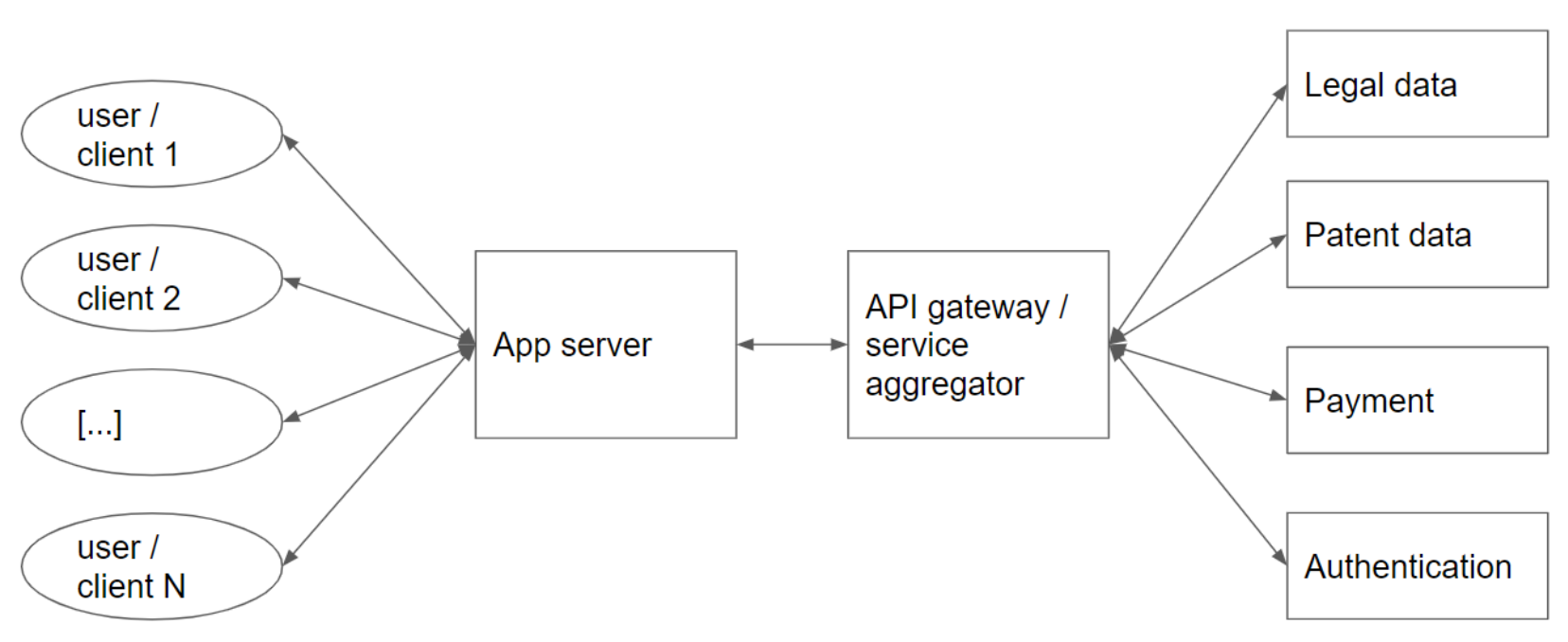
- "This system was designed for use in **libraries** and
- gave **users** access to **documents** that were
- stored in a number of **private databases**, such as
  - legal and
  - patent databases.

Data, databases

- **Payment** was required for **access** to these documents.
- The system had to:
  - manage the rights to these documents and
  - collect and account for access payments."

Services

Draft of architecture



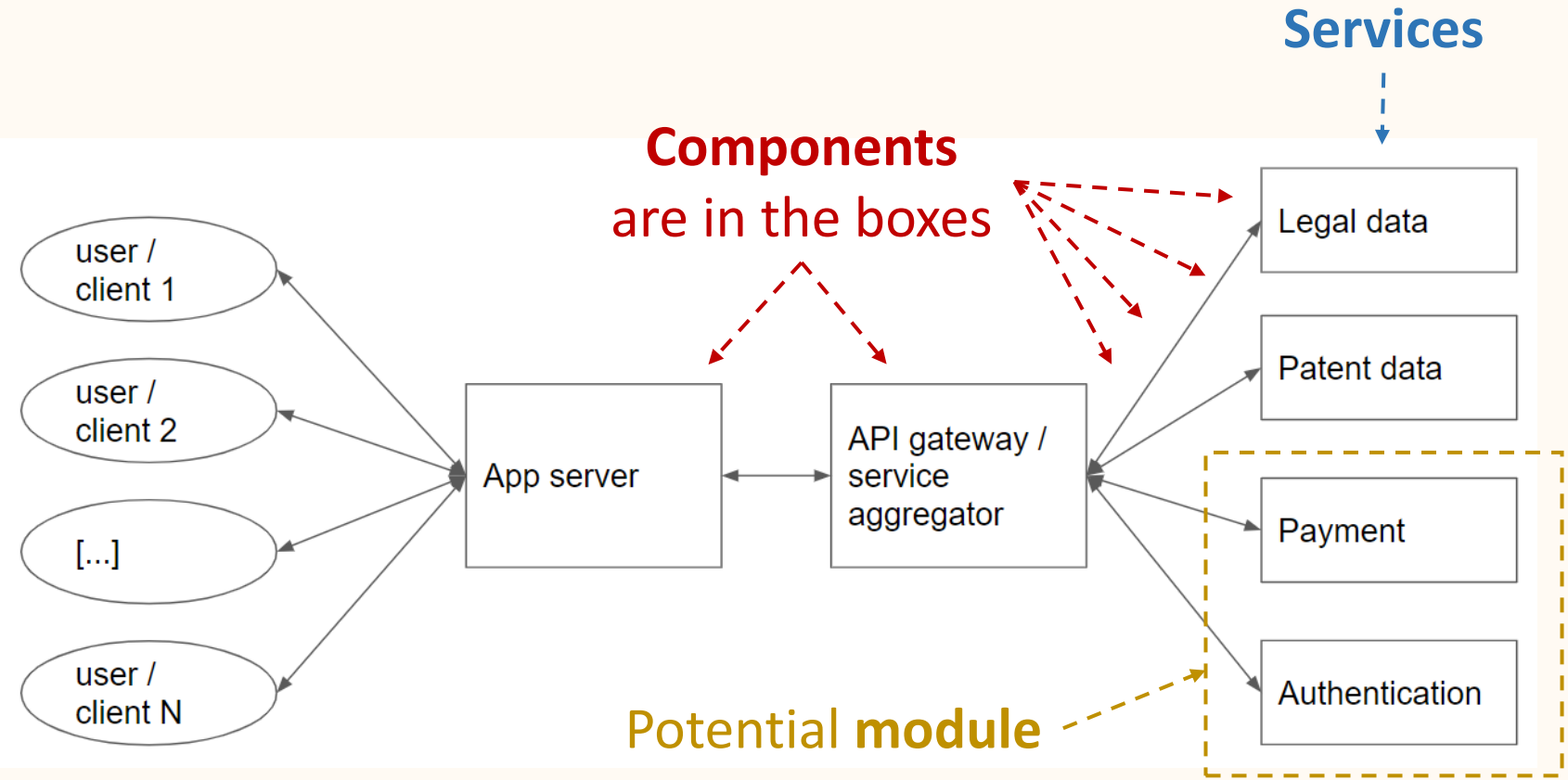
Outline

- Software architecture
- System quality attributes
- Design issues and trade-offs
- System decomposition
- Service-oriented architectures (SOA)

Terminology

- **Service**—coherent unit of functionality
  - May mean different things at different levels
- **Component**—named software unit offering service(s) to other components or end-users
  - Accessed by other components through an API
- **Module**—Named set of components
  - Components should have something in common
  - Example: may provide a set of related services

Terminology in context



Recall: relationships

- Depict connections between components
  - Could be structural, behavioral, or grouping
- Demonstrate how components are interrelated at *time of execution* (run-time)
- Important examples:
  - Composition (“has-a”)
  - Uses (makes calls to)
  - Generalization/realization (“is-a”)

Key relationships

- **Dependency**—A depends on B
  - A uses B in some way
  - Examples:
    - Calls to methods of B
    - Creation of B
    - Overriding behavior of B
  - Code changes in B may require code changes in A
  - Generic relationship (other relationships are specializations)

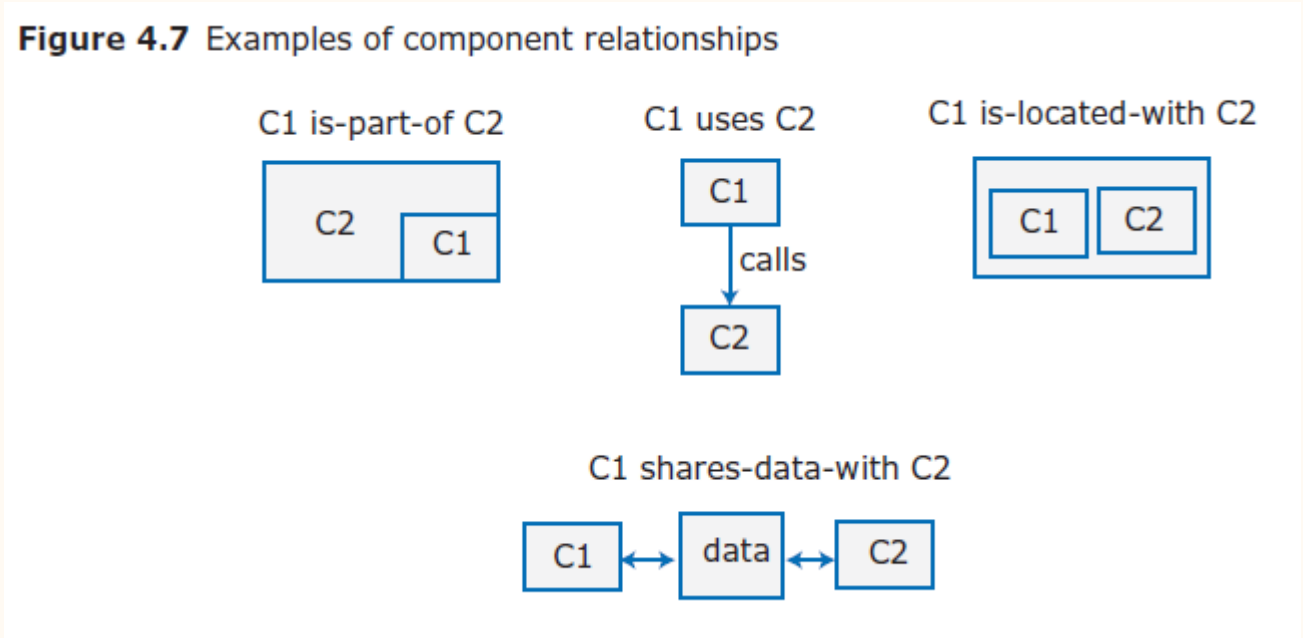
Key relationships

- **Composition**—A “has-a” B
  - If A is destroyed, so is its B (house/rooms, hand/fingers)
- **Generalization/Realization**—B “is-a” A
  - B implements/extends behavior of A
- **Aggregation**—A “knows about” B
  - B has own life-cycle, but A interacts with it

Key relationship characteristics

- How many?
  - 1-to-1
  - 1-to-many
  - Many-to-many
- What kind?
  - Sometimes relationships are labeled with kind

Relationships from book



What key relationships do you see here?

Summary

- Software architecture
- System quality attributes
- Design issues and trade-offs
- System decomposition
- Service-oriented architectures (SOA)