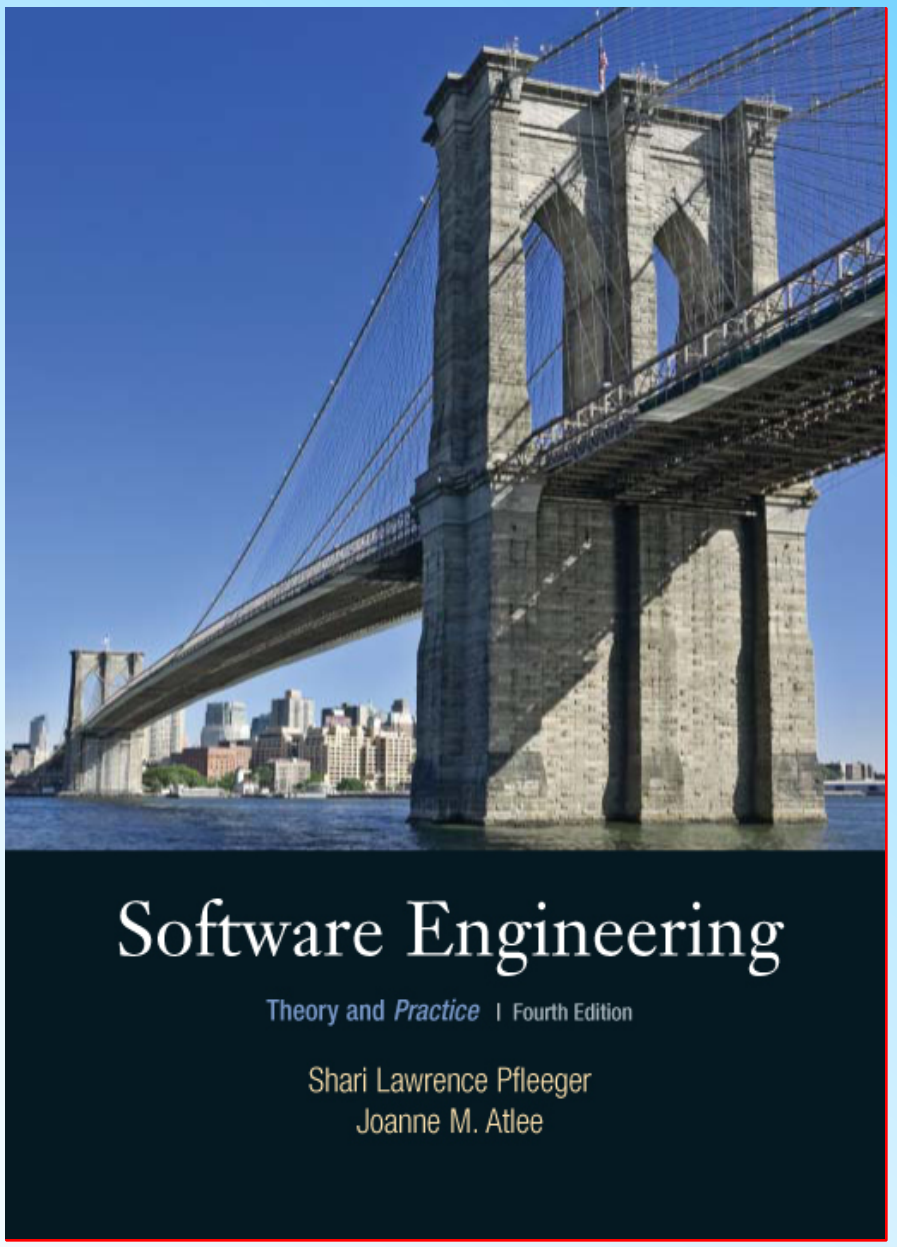
Chapter 5

Designing the Architecture

Shari L. Pfleeger
Joanne M. Atlee

4th Edition



Chapter 5 Objectives

- Examine different types of decomposition
- Compare competing designs
- Document the design
- Verify architecture meets the requirements

5.1 The Design Process

- **Design** is the creative process of figuring out how to implement all of the customer's requirements; the resulting plan is also called the design
- Early design decisions address the system's architecture
- Later design decisions address how to implement the individual units

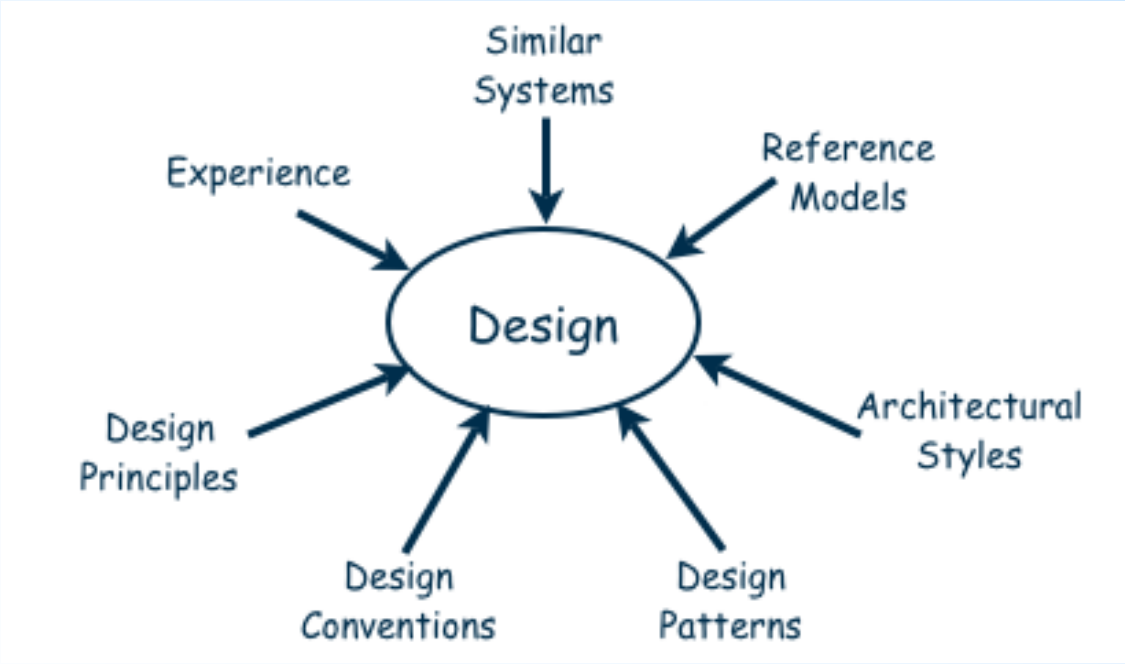
5.1 The Design Process

Design is a Creative Process

- Design is an intellectually challenging task
  - Numerous possibilities the system must accommodate
  - Nonfunctional design goals (e.g., ease of use, ease to maintain)
  - External factors (e.g., standard data formats, government regulations)
- We can improve our design by studying examples of good design
- Most design work is **routine design**, solve problem by reusing and adapting solutions from similar problems

5.1 The Design Process
Design is a Creative Process (continued)

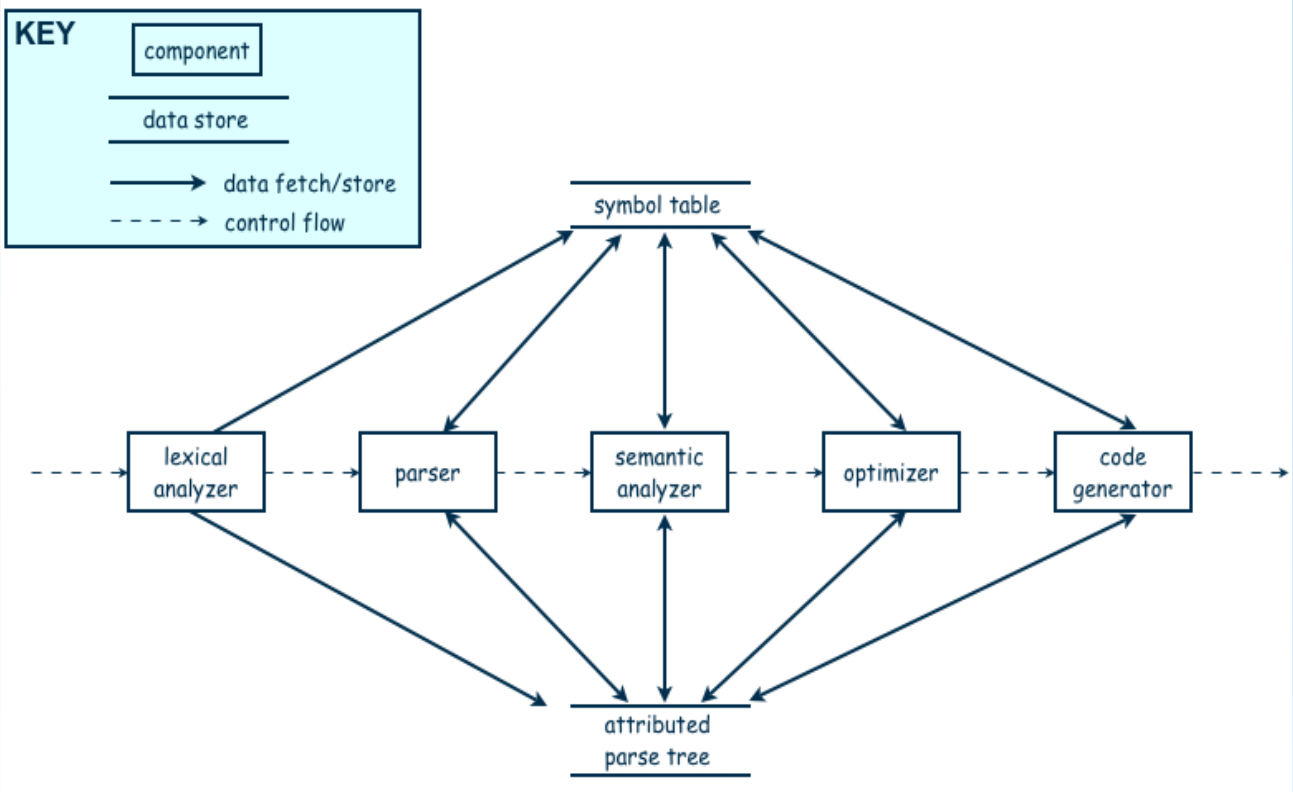
- Many ways to leverage existing solutions
  - Cloning: Borrow design/code in its entirety, with minor adjustments
  - Reference models: Generic architecture that suggests how to decompose the system



5.1 The Design Process

Design is a Creative Process (continued)

- Reference model for a compiler



5.1 The Design Process

Design is a Creative Process (continued)

- More typically, a reference model will not exist for the problem
- Software architectures have generic solutions too, referred to as **architectural styles**
  - Focusing on one architectural style can create problems
  - Good design is about selecting, adapting, and integrating several architectural design styles to produce the desired result

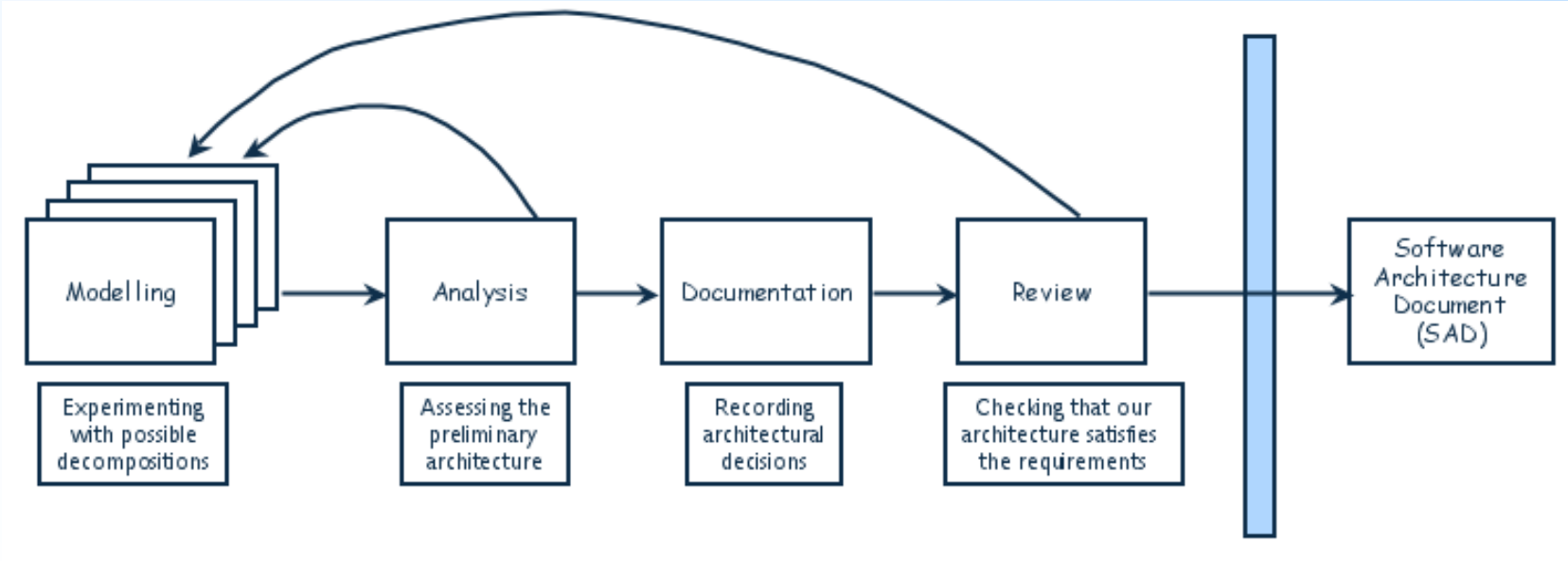
5.1 The Design Process
Design is a Creative Process (continued)

- Many tools for understanding options and evaluating chosen architecture, including:
  - Design patterns: generic solutions for making lower-level design decisions
  - Design convention or idiom: collection of design decisions and advice that, taken together, promotes certain design qualities
  - Innovative design: characterized by irregular bursts of progress that occur as we have flashes of insight
  - Design principles: descriptive characteristics of good design

5.1 The Design Process

Design Process Model

- Designing software system is an iterative process
- The final outcome is the software architecture document (SAD)



5.2 Modeling Architectures

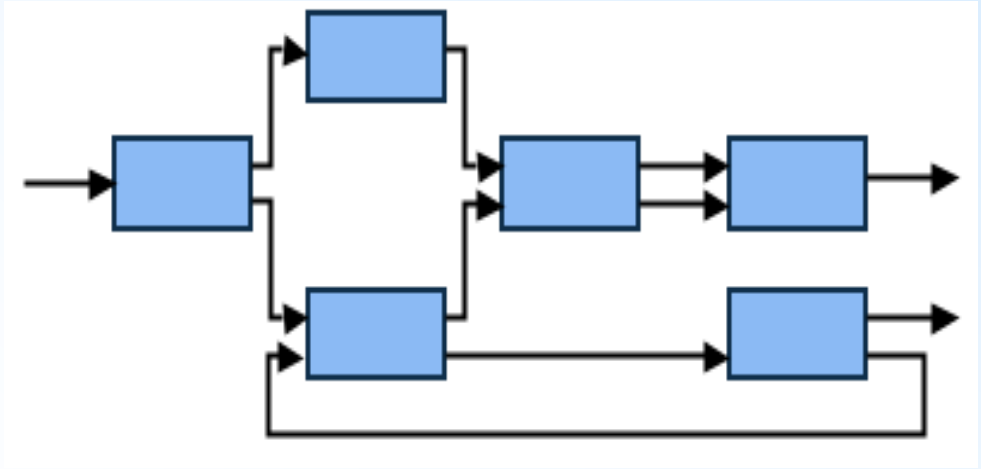
- Collection of models helps to answer whether the proposed architecture meets the specified requirements
- Six ways to use the architectural models:
  - to understand the system
  - to determine amount of reuse from other systems and the reusability of the system being designed
  - to provide blueprint for system construction
  - to reason about system evolution
  - to analyze dependencies
  - to support management decisions and understand risks

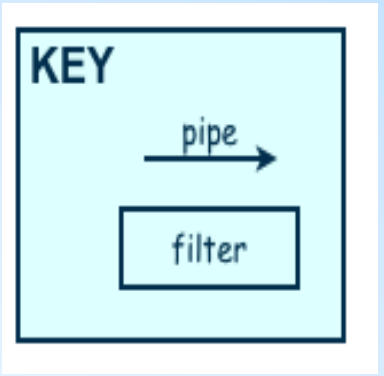
5.4 Architectural Styles and Strategies

- Pipes-and-Filter
- Client-Server
- Peer-to-Peer
- Publish-Subscribe
- Repositories
- Layering

5.4 Architectural Styles and Strategies
Pipes-and-Filter

- The system has
  - Streams of data (pipe) for input and output
  - Transformation of the data (filter)



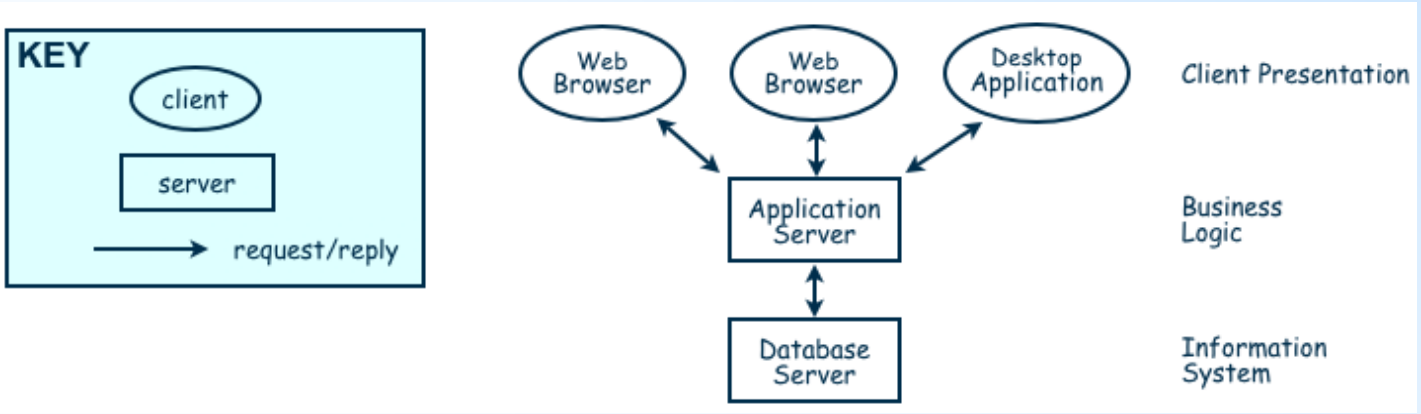


5.4 Architectural Styles and Strategies
Pipes-and-Filter (continued)

- Several important properties
  - The designer can understand the entire system's effect on input and output as the composition of the filters
  - The filters can be reused easily on other systems
  - System evolution is simple
  - Allow concurrent execution of filters
- Drawbacks
  - Encourages batch processing
  - Not good for handling interactive application
  - Duplication in filters functions

5.4 Architectural Styles and Strategies
Client-Server

- Two types of components:
  - Server components offer services
  - Clients access them using a request/reply protocol
- Client may send the server an executable function, called a callback
  - The server subsequently calls under specific circumstances



5.4 Architectural Styles and Strategies

Sidebar 5.3 The World Cup Client-Server System

- Over one month in 1994, the World Cup soccer matches were held in the United States. Design system issues:
  - 24 teams played 52 games
  - nine different cities that spanned four time zones
  - results of each game were recorded and disseminated to the press and to the fans
  - To deter violence among the fans, the organizers issued and tracked over 20,000 identification passes
- This system required both central control and distributed functions. Thus, a client-server architecture seemed appropriate.
- The system that was built included a central database, located in Texas, for ticket management, security, news services, and Internet links. This server also calculated games statistics and provided historical information, security photographs, and clips of video action.
- The clients ran on 160 Sun workstations that were located in the same cities as the games and provided support to the administrative staff and the press

5.4 Architectural Styles and Strategies
Peer-to-Peer (P2P)

- Each component acts as its own process and acts as both a client and a server to other peer components.
- Any component can initiate a request to any other peer component.
- Characteristics
  - Scale up well
  - Increased system capabilities
  - Highly tolerant of failures
- Examples: Napster and Freenet

5.4 Architectural Styles and Strategies
Sidebar 5.4 Napster's P2P Architecture

- Peers are typically users' desktop computer systems running general-purpose computing applications (email, word processors, Web browsers, etc.)
  - Many user systems do not have stable Internet protocol (IP) addresses
  - Not always available to the rest of the network
  - Most users are not sophisticated; they are more interested in content than in the network's configuration and protocols
  - Great variation in methods for accessing the network, from slow dial-up lines to fast broadband connections
- Napster's sophistication comes from its servers, which organize requests and manage content, with actual content provided by users, shared from peer to peer, and the sharing goes to other (anonymous) users, not to a centralized file server
- If the file content changes frequently, sharing speed is key, file quality is critical, or one peer needs to be able to trust another, a centralized server architecture may be more appropriate

5.4 Architectural Styles and Strategies
Publish-Subscribe

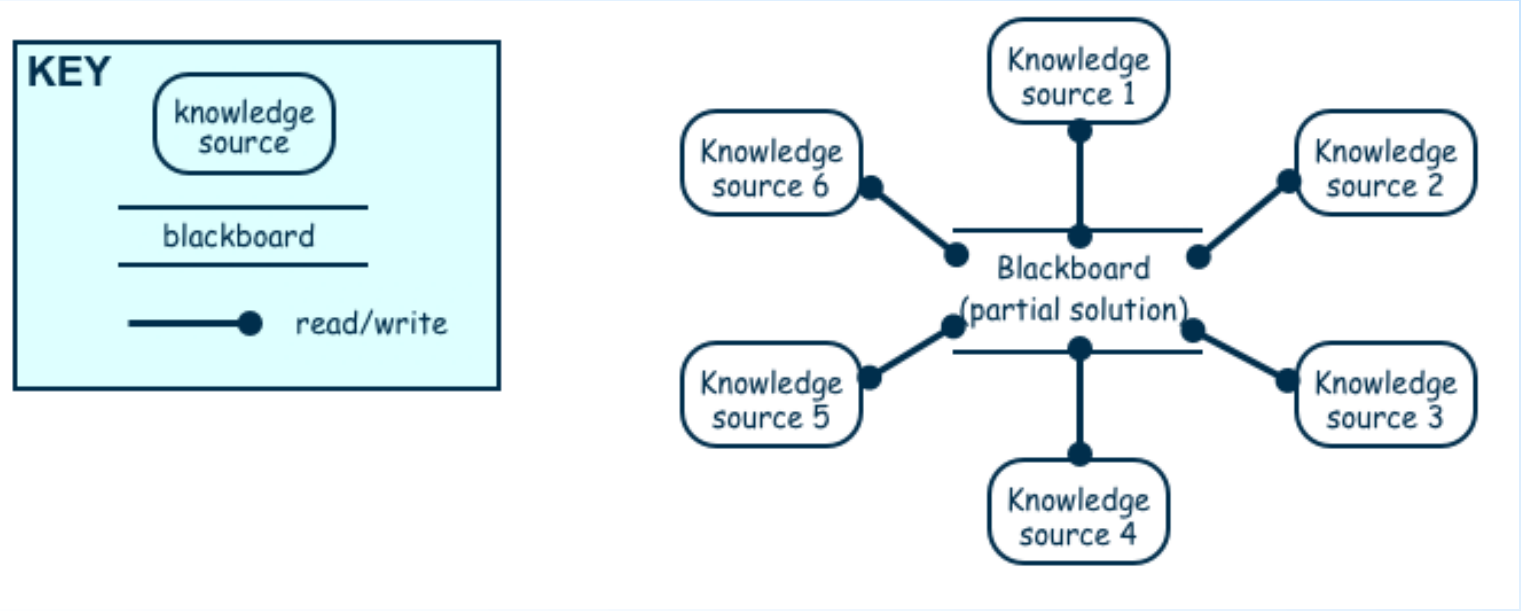
- Components interact by broadcasting and reacting to events
  - Component expresses interest in an event by subscribing to it
  - When another component announces (publishes) that event has taken place, subscribing components are notified
  - Implicit invocation is a common form of publish-subscribe architecture
    - Registering: subscribing component associates one of its procedures with each event of interest (called the procedure)
- Characteristics
  - Strong support for evolution and customization
  - Easy to reuse components in other event-driven systems
  - Need shared repository for components to share persistent data
  - Difficult to test

5.4 Architectural Styles and Strategies
Repositories

- Two components
  - A central data store
  - A collection of components that operate on it to store, retrieve, and update information
- The challenge is deciding how the components will interact
  - A traditional database: transactions trigger process execution
  - A blackboard: the central store controls the triggering process
  - Knowledge sources: information about the current state of the system's execution that triggers the execution of individual data accessors

5.4 Architectural Styles and Strategies
Repositories (continued)

- Major advantage: openness
  - Data representation is made available to various programmers (vendors) so they can build tools to access the repository
  - But also a disadvantage: the data format must be acceptable to all components



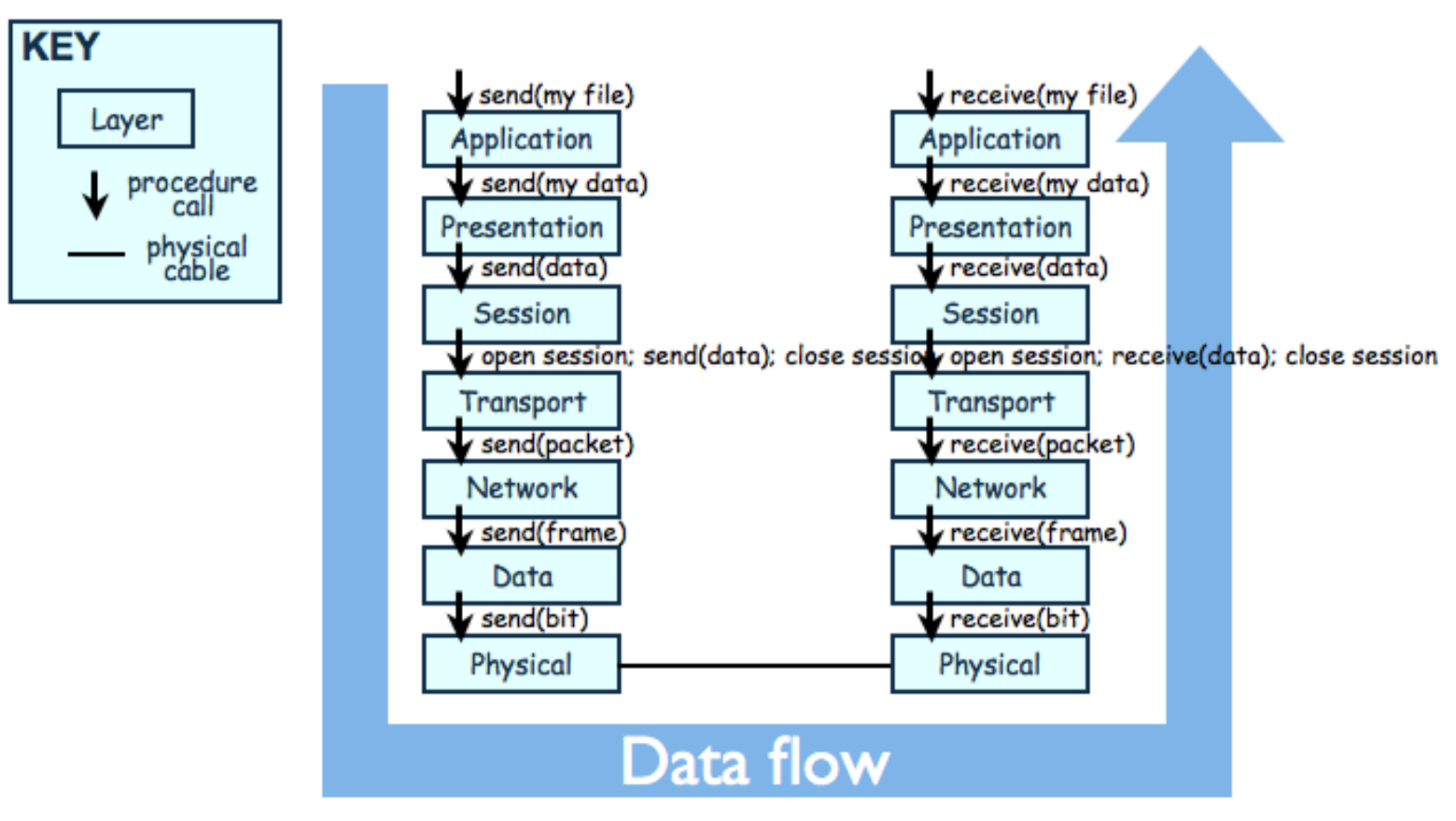
5.4 Architectural Styles and Strategies
Layering

- Layers are hierarchical
  - Each layer provides service to the one outside it and acts as a client to the layer inside it
  - Layer bridging: allowing a layer to access the services of layers below its lower neighbor
- The design includes protocols
  - Explain how each pair of layers will interact
- Advantages
  - High levels of abstraction
  - Relatively easy to add and modify a layer
- Disadvantages
  - Not always easy to structure system layers
  - System performance may suffer from the extra coordination among layers

5.4 Architectural Styles and Strategies

Example of Layering System

- The OSI Model

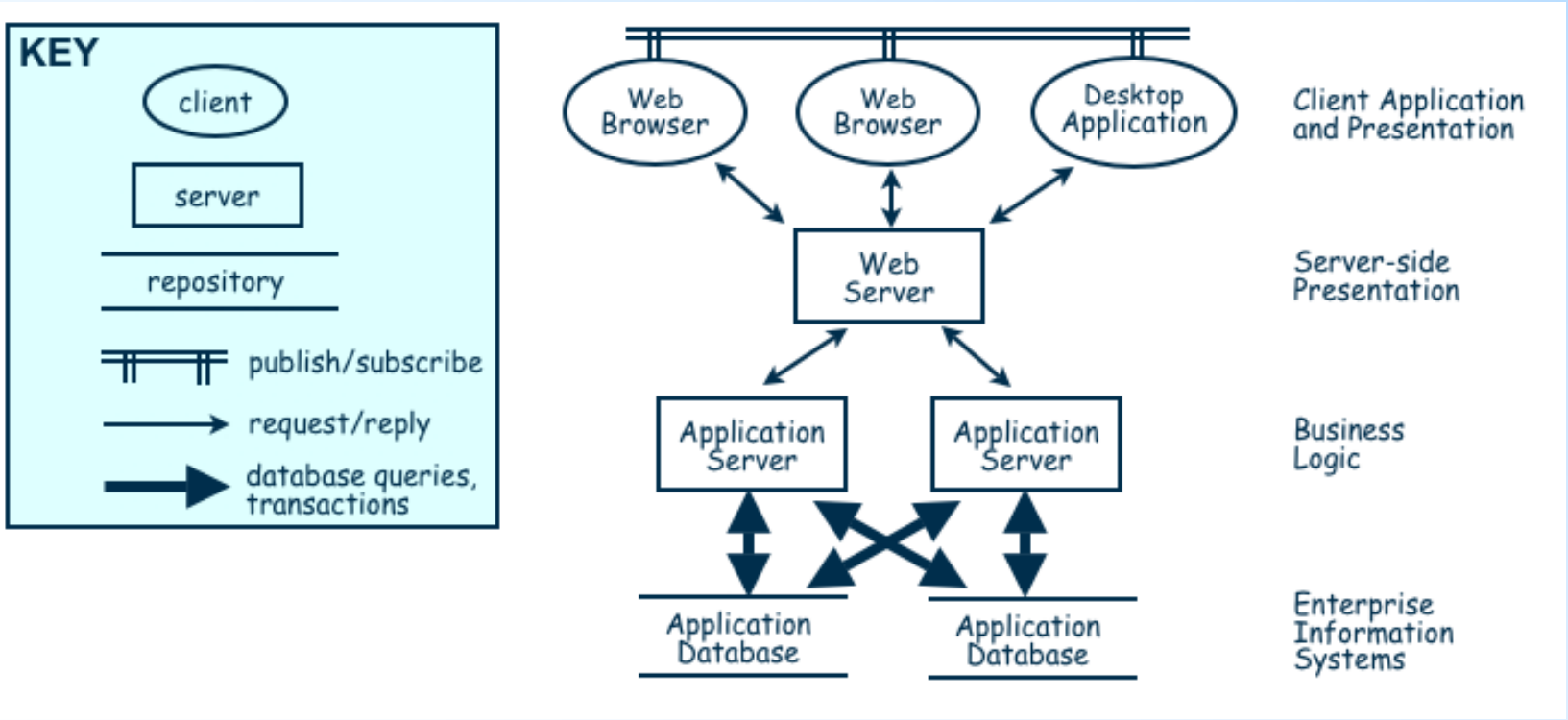


5.4 Architectural Styles and Strategies
Combining Architectural Styles

- Actual software architectures rarely based on purely one style
- Architectural styles can be combined in several ways
  - Use different styles at different layers (e.g., overall client-server architecture with server component decomposed into layers)
  - Use mixture of styles to model different components or types of interaction (e.g., client components interact with one another using publish-subscribe communications
- If architecture is expressed as collection of models, documentation must be created to show relation between models

5.4 Architectural Styles and Strategies

Combination of Publish-Subscribe, Client-Server, and Repository Architecture Styles



5.5 Achieving Quality Attributes

- Architectural styles provide general beneficial properties. To support specific quality attribute tactics are utilized:
  - Modifiability
  - Performance
  - Security
  - Reliability
  - Robustness
  - Usability
  - Business goals

5.5 Achieving Quality Attributes
Modifiability

- Design must be easy to change
- Two classifications of affected software units:
  - Directly affected
  - Indirectly affected
- Directly affected units' responsibilities change to accommodate a system modification
- Indirectly affected units' responsibilities do not change, but implementations must be revised

5.5 Achieving Quality Attributes
Modifiability (continued)

- Tactics for minimizing the number of software units affected by a change focus on clustering the anticipated changes:
  - Anticipate expected changes: Identify design decisions that are most likely to change, and encapsulate each in its own software unit
  - Cohesion: Keeping software units highly cohesive increases the chances that a change to the system's responsibilities is confined to the few units that are assigned those responsibilities
  - Generality : The more general the software units, the more likely change can be accommodated by modifying a unit's inputs rather than modifying the unit itself

5.5 Achieving Quality Attributes
Modifiability (continued)

- Tactics for minimizing the impact on indirectly affected units focus on reducing dependencies
  - Coupling: Lowering coupling reduces the likelihood that a change to one unit will ripple to other units
  - Interfaces: If a unit interacts with other units only through their interfaces changes to one unit will not spread beyond the unit's boundary unless its interface changes
  - Multiple interfaces: A unit modified to provide new data or services can offer them using a new interface to the unit without changing any of the unit's existing interfaces

5.5 Achieving Quality Attributes
Sidebar 5.5 Self-managing Software

- In response to increasing demands that systems be able to operate optimally in different and sometimes changing environments, the software community is starting to experiment with self-managing software
  - Also referred to as autonomic, adaptive, dynamic, selfconfiguring, self-optimizing, self-healing, context-aware
- The essential idea is the same: the software system monitors its environment or its own performance, and changes its behavior in response to changes that it

5.5 Achieving Quality Attributes

Sidebar 5.5 Self-managing Software (continued)

- Some examples of sensor changes:
  - Change the input sensors used, such as avoiding vision-based sensors when sensing in the dark
  - Change the Web servers that are queried, based on the results and performance of past queries
  - Move running components to different processors to balance processor load or to recover from a processor failure
- Obstacles to building self-managing software:
  - Few architectural styles
  - Monitoring nonfunctional requirements
  - Decision making

5.5 Achieving Quality Attributes
Performance

- Performance attributes describe constraints on system speed and capacity:
  - Response time: How fast does our software respond to requests?
  - Throughput: How many requests can it process per minute?
  - Load: How many users can it support before response time and throughput start to suffer?

5.5 Achieving Quality Attributes
Performance

- Tactics for improving performance include:
  - Improve utilization of resources
  - Manage resource allocation more effectively
    - First-come/first-served: Requests are processed in the order in which they are received
    - Explicit priority: Requests are processed in order of their assigned priorities
    - Earliest deadline first: Requests are processed in order of their impending deadlines
  - Reduce demand for resources

5.5 Achieving Quality Attributes
Security

- Two key architectural characteristics particularly relevant to security: immunity and resilience
- **Immunity**: ability to thwart an attempted attack
  - The architecture encourages immunity by:
    - Ensuring all security features are included in the design
    - Minimizing exploitable security weaknesses
- **Resilience**: ability to recover quickly and easily from an attack
  - The architecture encourages resilience by:
    - Segmenting functionality to contain attack
    - Enabling the system to quickly restore functionality

5.5 Achieving Quality Attributes
Reliability

- A software system is reliable if it correctly performs its required functions under assumed conditions
  - Is the software internally free of errors?
- A **fault** is the result of human error, compared to a **failure**, which is an observable departure from required behavior
  - Software is made more reliable by preventing or tolerating faults

5.5 Achieving Quality Attributes
Reliability (continued)

- **Passive fault detection**: wait until fault occurs during execution
- **Active fault detection**: periodically check for symptoms or try to anticipate when failures will occur
- **Exceptions**: situations that cause the system to deviate from its desired behavior
- Include **exception handling** in design to handle exception and return system to acceptable state
- Typical exceptions include:
  - Failing to provide a service
  - Providing the wrong service
  - Corrupting data
  - Violating a system invariant (e.g.; security property)
  - Deadlocking

5.5 Achieving Quality Attributes
Reliability (continued)

- N-version programming
  - If two functionally equivalent systems are designed by two different design teams at two different times using different techniques, the chance of the same fault occurring in both implementations is very small
  - N-version programming has been shown to be less reliable than originally thought, because many designers learn to design in similar ways, using similar design patterns and principles

5.5 Achieving Quality Attributes
Reliability (continued)

- **Fault recovery**: handling fault immediately to limit damage
- Fault recovery tactics:
  - Undoing transactions: manage a series of actions as a single transaction that are easily undone if a fault occurs midway through the transaction
  - Checkpoint/rollback: software records a checkpoint of current state; rolls back to that point if system gets in trouble
  - Backup: system automatically substitutes faulty unit with backup
  - Degraded service: returns to previous state, offers degraded version of the service
  - Correct and continue: detects the problem and treats the symptoms
  - Report: system returns to its previous state and reports the problem to an exception-handling unit

5.5 Achieving Quality Attributes
Sidebar 5.6 The Need for Safe Design

- From 1986 to 1997, over 450 reports filed with the U.S. Food and Drug Administration (FDA) detailing software defects in medical devices, 24 of which led to death or injury
  - Numbers may be greater based on time to file report
- The FDA established a software forensics unit in 2004 after noticing that medical device makers were reporting more and more software-based recalls
- Software designers must see directly how their products will be used
- Then designers can build in preventative measures to ensure their products are not misused

5.5 Achieving Quality Attributes

Robustness

- A system is **robust** if it includes mechanisms for accommodating or recovering from problems in the environment or in other unit
- Mutual suspicion: each software unit assumes that the other units contain faults
- Robustness tactics differ from reliability tactics
- Recovery tactics are similar:
  - Rollback to checkpoint state
  - Abort a transaction
  - Initiate a backup unit
  - Provide reduced service
  - Correct symptoms and continue processing
  - Trigger an exception

5.5 Achieving Quality Attributes
Usability

- Usability reflects the ease in which a user is able to operate the system
  - User interface should reside in its own software unit
  - Some user-initiated commands require architectural support
  - There are some system-initiated activities for which the system should maintain a model of its environment

5.5 Achieving Quality Attributes
Business Goals

- Business Goals are quality attributes the system is expected to exhibit (e.g., minimizing the cost of development and time to market)
  - Buy vs. Build
    - Save development time, money
    - More reliable
    - Existing components create constraints; vulnerable to supplier
  - Initial development vs. maintenance costs
    - Save money by making system modifiable
    - Increased complexity may delay release; lose market to competitors
  - New vs. known technologies
    - Acquiring expertise costs money, delays product release
    - Either learn how to use the new technology or hire new personnel
    - Eventually, we must develop the expertise ourselves

5.8 Documenting Software Architectures

- System's architecture is vital to overall development and serves as the basis on decisions for:
  - Design
  - Quality assurance
  - Project management
- The SAD serves as the repository for design information and includes:
  - System overview
  - Views
  - Software units
  - Analysis data and results
  - Design rationale
  - Definitions, glossary, acronyms

5.8 Documenting Software Architectures
Mappings among Views

- Structure of the system and intended measured attributes determine number and type of views to include in SAD
  - should at least include decomposition and execution view
- Design is collection of views; must show how views relate to one another

5.8 Documenting Software Architectures
Documenting Rationale

- **Document rationale**: outlining critical issues and trade-offs
- When to document the rationale behind decision:
  - Significant time spent on decision
  - Decision is critical
  - Decision is counterintuitive
  - Costly to change decision

5.9 Architecture Design Review

- Design review is an essential part of engineering practice
- SAD quality is evaluated in two ways:
  - **Validation**: making sure the design satisfies all of the customer's requirements (i.e., is this the right system?)
  - **Verification**: ensuring the design adheres to good design principles (i.e., are we building the system right?)

5.9 Architecture Design Review
Validation

- Several key people included in review:
  - The analyst(s) who helped define the system requirements
  - The system architect(s)
  - The program designer(s) for this project
  - A system tester
  - A system maintainer
  - A moderator
  - A recorder
  - Other interested developers not otherwise involved in this project

5.9 Architecture Design Review
Verification

- Judge whether it adheres to good design principles:
  - Is the architecture modular, well structured, and easy to understand?
  - Can we improve the structure and understandability of the architecture?
  - Is the architecture portable to other platforms?
  - Are aspects of the architecture reusable?
  - Does the architecture support ease of testing?
  - Does the architecture maximize performance, where appropriate?
  - Does the architecture incorporate appropriate techniques for handling faults and preventing failures?
  - Can the architecture accommodate all of the expected design changes and extensions that have been documented?

5.9 Architecture Design Review
Verification (continued)

- **Active design review**: exercise the design document by using is in ways the developers will use the final document in practice
- **Passive review process**: reading the documentation and looking for problems

5.11 Information System Example
Piccadilly System

- What might be a suitable architecture for the Piccadilly systems?
- Key components
  - A repository of information
  - Address multiple heterogeneous queries
- A typical reference architecture for an information system
  - n-tiered client-server architecture

5.12 Real-Time Example
Ariane-5 Failure

- Inquiry found that the Ariane program had a "culture...of only addressing random hardware failures" and assuming the software was correct
- Hardware failures are independent of one another
- Software faults tend to be logical
  - All redundant components will have the same faults
- Redundancy in Ariane-5 is likely to recover only from hardware failures

5.13 What This Chapter Means For You

- Systems need to be designed based on carefully expressed requirements
- Design begins with a high-level architecture, where architectural decisions are based not only on system functionality and required constraints but also on desirable attributes and the long-term intended use of the system (including product lines, reuse, and likely modification)
- Keep in mind several characteristics of good architecture as you go, including appropriate user interfaces, performance, modularity, security, and fault tolerance
- The goal is not to design the ideal software architecture for a system, because such an architecture might not even exist. Rather, the goal is to design an architecture that meets all of the customer's requirements while staying within the cost and schedule constraints