

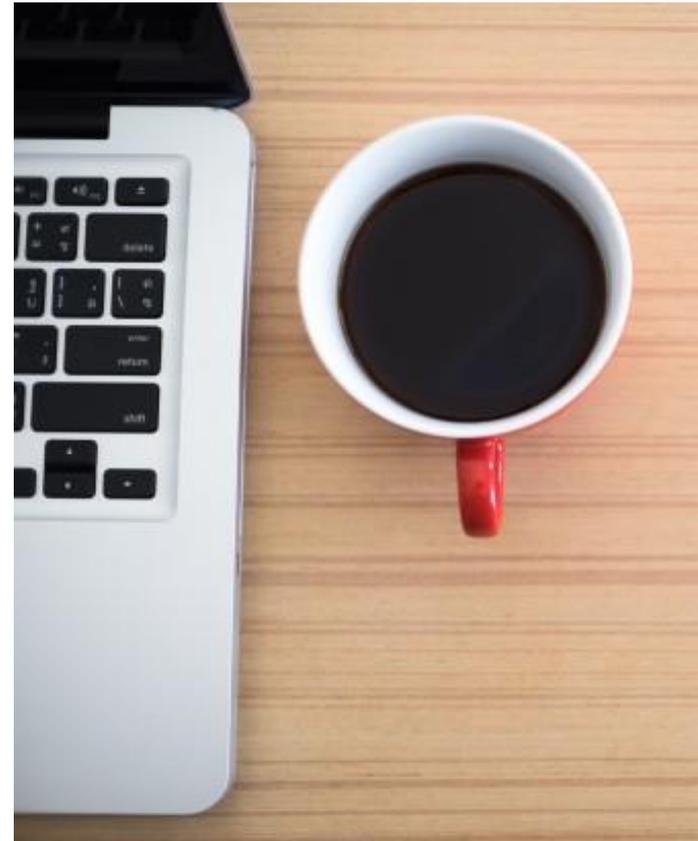


# Course Topics

- ~~Introduction~~
- ~~Software Process Models~~
- ~~Requirements Engineering~~
- Modeling
- Programming Languages
- Software Construction Techniques
- Testing
- Project Management
- Refactoring
- Ethical Issues

# Lecture Objectives

- ✓ Architectural design
- ✓ Architecture Characteristics
- ✓ 4+ 1 Architectural Views
- ✓ Architectural Patterns



# Introduction to Design



- Once the requirements of a project are understood, the transformation of requirements into a design begins.
  - This a difficult step that involves the transformation of a set of intangible (the requirements) into another set of intangible (the design).
  - Software design details with how the software is to be structured – that is, what its components are and how these components are related to each other.
- 

# Introduction to Design



- For a large system, it usually makes sense to divide the design phases into two parts:
- *Architectural design phase*
- *Detailed design phase*

# Introduction to Design



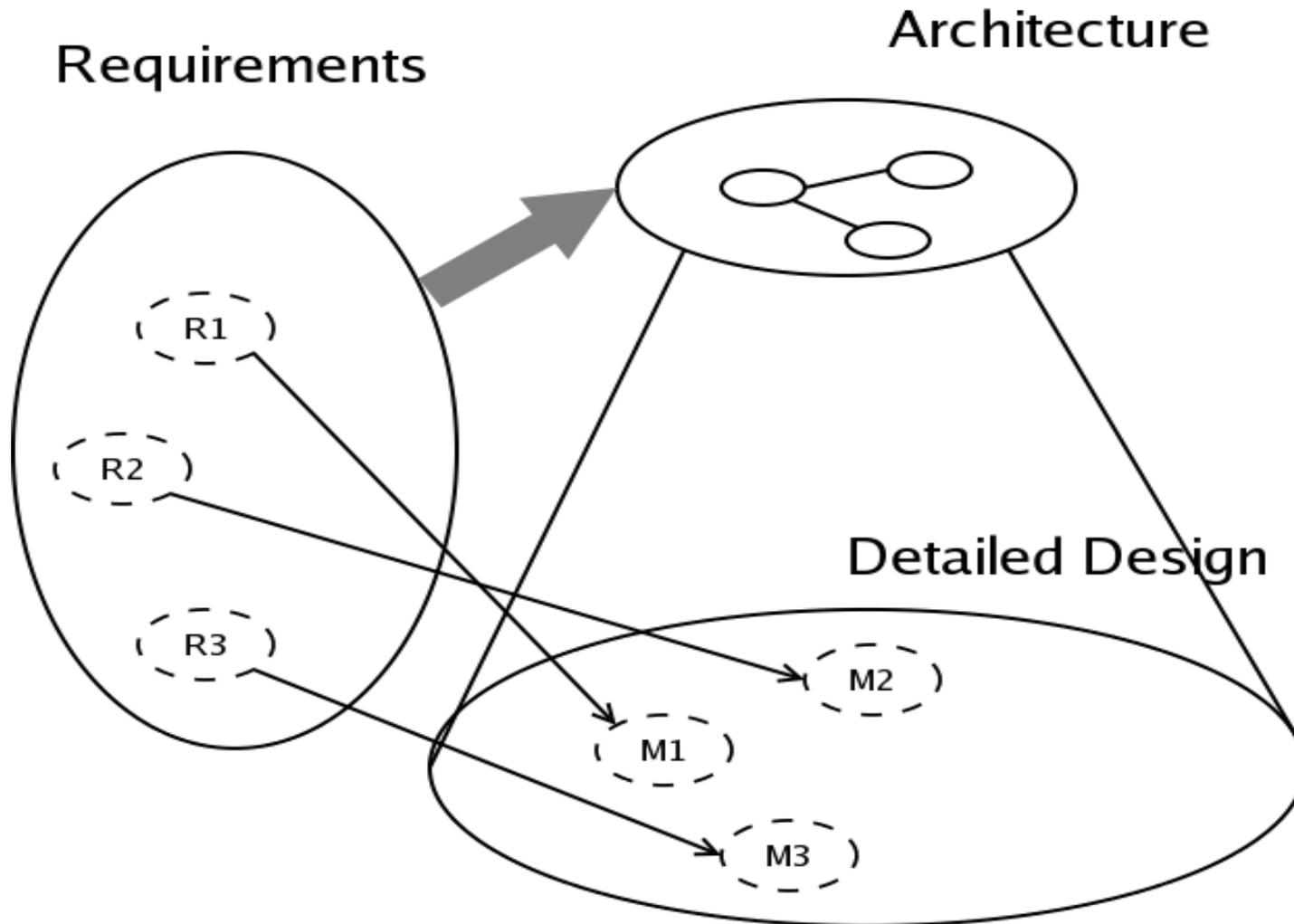
- Architectural design phase – This a high-level overview of the system.
  - The main components are listed as well as properties external to the components and relationships among components.
  - The functional and nonfunctional requirements along with technical consideration provide most of the drive for the architecture.
- 

# Introduction to Design

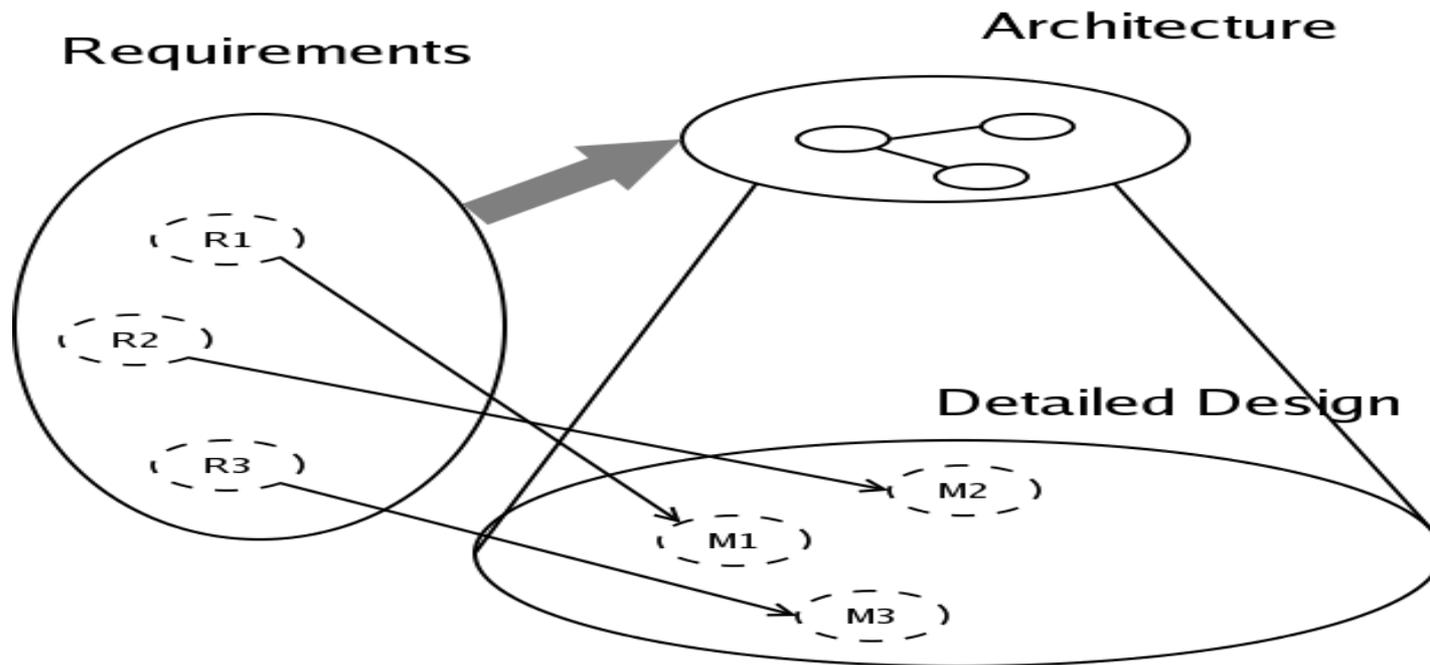


- Detailed design phase – components are decomposed to a much finer level of details.
- The architecture and the functional requirements drive this phase.
- The architecture provides general guidance and all functional requirements have to be addressed by at least one module in the detailed design.

# Relationship between Architecture and Design



# Relationship between Architecture and Design



**Most influential requirements may be nonfunctional requirements, such as performance and maintainability.**

# Relationship between Architecture and Design



- Ideally there is a one-to-one mapping between each functional requirement and a module in the detailed design.
- The architecture drives the detailed design, with the mapping being ideally from one architectural component to several detailed modules.

# Relationship between Architecture and Design



- Smaller systems may get away with not having an explicit architecture, although it is useful in almost all cases.
- In traditional software processes, the ideal is for the design to be created and documented up to the lowest level of detail possible,
  - with the programmers doing mainly translation of that design into actual code.

# Architectural Design



- What is **software architecture**?
  - The software architecture of a system specifies its basic **structure**.
  - The design process for identifying the **sub-systems** making up a system and the framework for sub-system control and communication is architectural design.
  - The output of this design process is a description of the software architecture.

# System Structuring

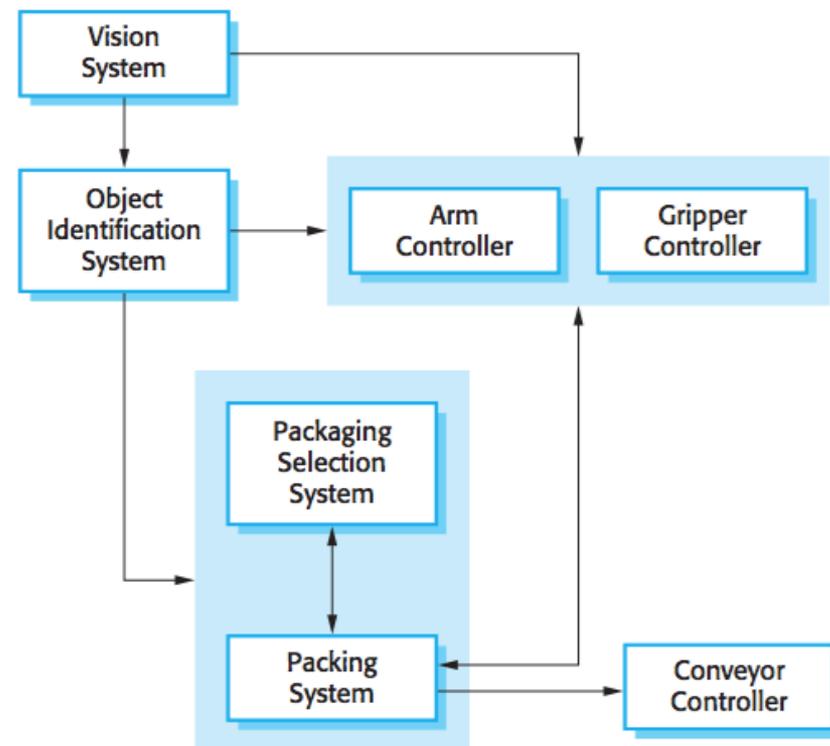


- Concerned with decomposing the system into interacting sub-systems.
- The architectural design is normally expressed as a **block diagram** presenting an overview of the system structure.
- More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed.

# Block Diagrams

- Very abstract
  - they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
- However, useful for communication with stakeholders and for project planning.

- Packing Robot System



# Architectural Design



- There are several important points to note about the architecture of a system:
  1. Every system has an architecture.
    - Whether you make it explicit or not, whether you document it or not, the system has an architecture.
  2. There could be more than one structure.
    - For large systems, and even many small ones, there is more than one important way the system is structured.
    - We need to be aware of all those structures, and document them with several views.

# Architectural Design



3. Architecture deals with properties external to each module.
  - At the architectural level, we should think about the important modules and how they interact with other modules.
  - The focus is on the interfaces among modules rather than details concerning the internals of each module.

# Advantages of Explicit Architecture



- Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders.
- System analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible.
- Large-scale reuse
  - The architecture may be reusable across a range of systems.

# Architecture Design Decisions (Architecture Characteristics)



## ■ Performance

- Localize critical operations and minimize communications. Use large rather than fine-grain components.

## ■ Security

- Use a layered architecture with critical assets in the inner layers.

## ■ Safety

- Localize safety-critical features in a single component or small number of components.

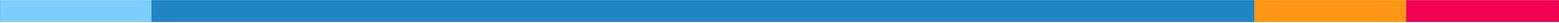
## ■ Availability

- Include redundant components and mechanisms for fault tolerance.

## ■ Maintainability

- Use fine-grain, self-contained components, replaceable components.
- 

# Architectural *Conflicts*



- ▶ Using large components improves *performance* but reduces *maintainability*.
  - ▶ Introducing redundant data improves *availability* but makes *security* more difficult.
  - ▶ Localizing *safety-related* features usually means more communication so degraded *performance*.
- 

# Architectural Views

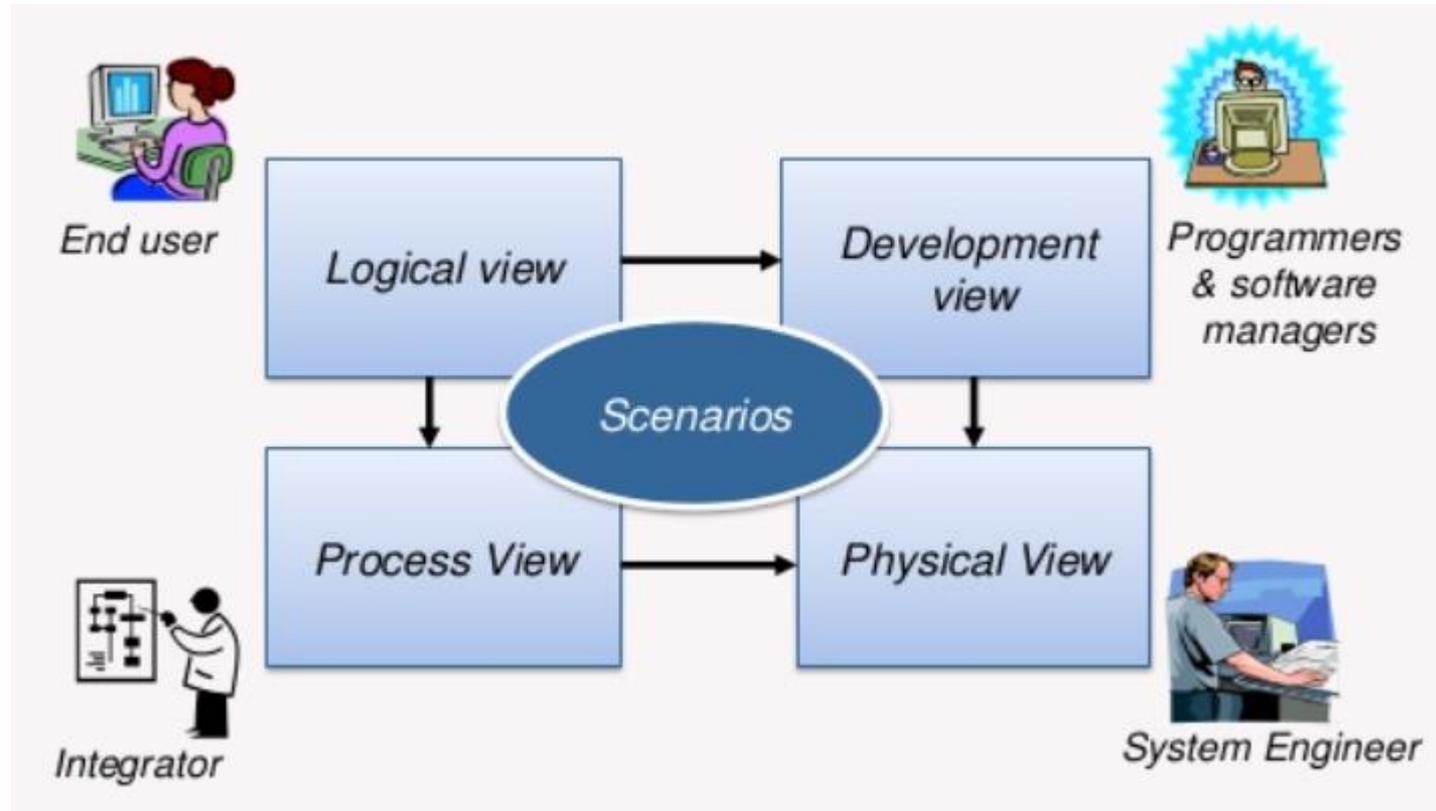


- Each architectural model only shows one view or perspective of the system. It might show
  - how a system is decomposed into modules
  - how the run-time processes interact
  - different ways in which system components are distributed across a network.
- For both design and documentation, you usually need to present multiple views of the software architecture.

# Architectural Views



- **4+1 view** model of software architecture:
    - ▶ **logical** view, which shows the key abstractions in the system as **objects or object classes** (relate the system requirements to entities in this logical view).
    - ▶ A **process** view, which shows how, at run-time, the system is composed of **interacting processes** (useful for making judgments about non-functional system characteristics).
    - ▶ A **development** view, which shows how the software is decomposed for **development** (useful for managers and programmers).
    - ▶ A **physical** view, which shows the **system hardware** and how software components are distributed across the processors in the system (useful for system deployment).
    - ▶ Related using use cases or scenarios (+1).
- 



# Architectural patterns

# Architecture Patterns



- Stylized, abstract description of good practice.
- Tried and tested in different systems and environments
  - Successful in previous systems
- Includes information on when it is and is not appropriate to use that pattern.
- Includes information on the pattern's strengths and weaknesses.

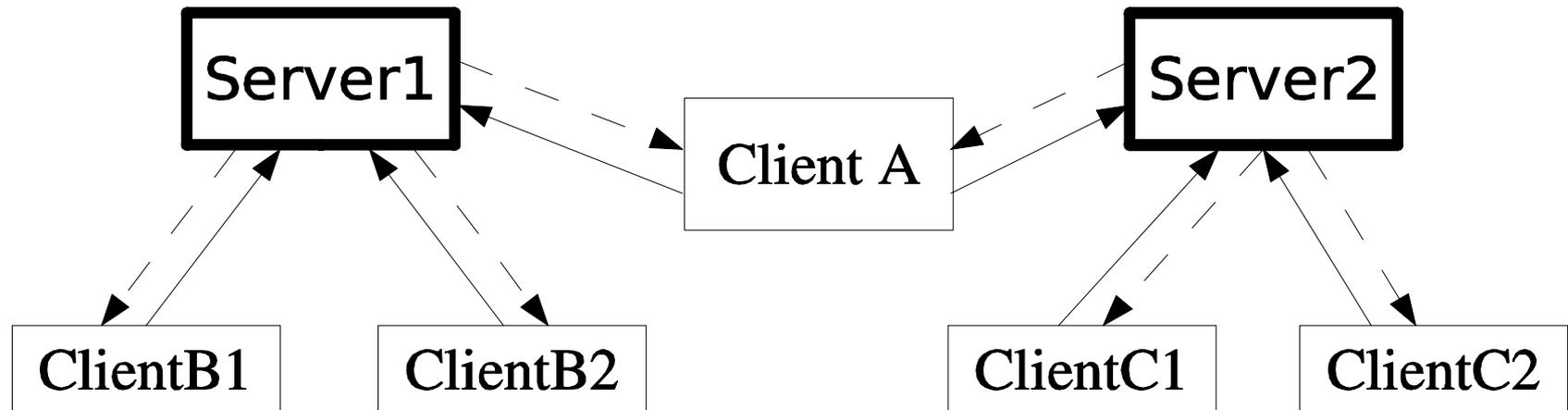
# Architecture Patterns Covered



- Client Server Architecture
- Layered Architecture
- Repository
- Event Driven

# Client-Server Architecture

- Application split into client components and server components.
- Client may connect to more than one server (servers are usually independent).



# Client-Server Architecture



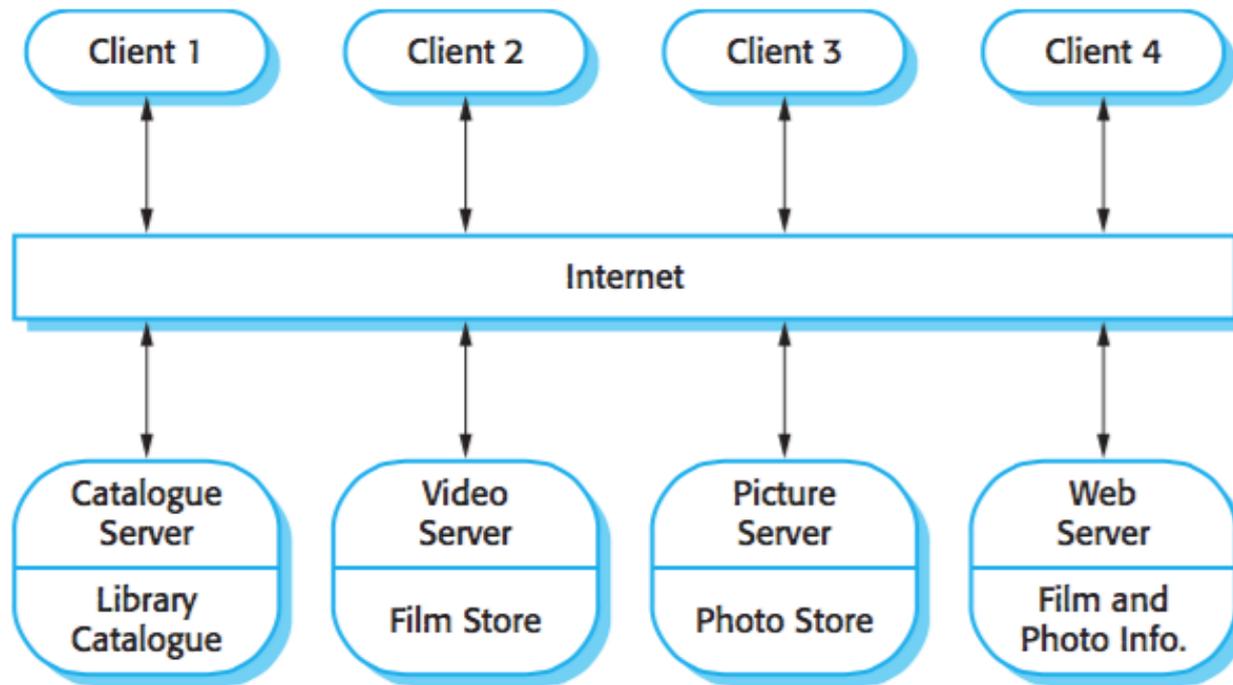
- An architecture showing a clear demarcation between clients and servers, which reside on different nodes in a network.
- Components interact through basic networking protocols.
- Usually there will be many clients accessing the same server.

# Client-Server Architecture

Name	Client-server
Description	In a client-server architecture, the system is presented as a set of services, with each service delivered by a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.13 is an example of a film and video/DVD library organized as a client-server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure and so is susceptible to denial-of-service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. Management problems may arise if servers are owned by different organizations.

# Client-Server Architecture

- Below architecture is a multi-user, web-based system for providing a film and photograph library.



# Layered Architecture

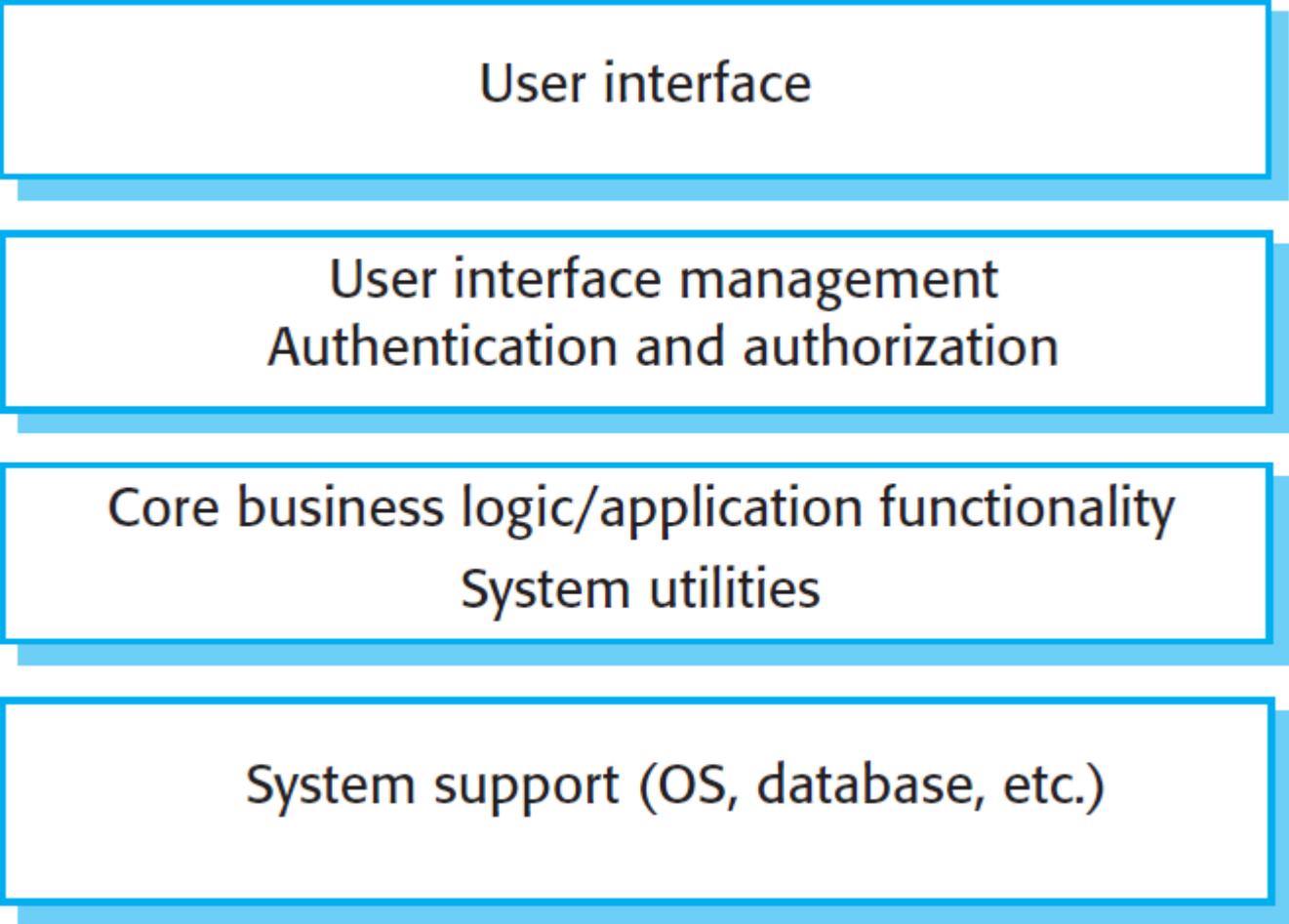


- The layered architecture aims at achieving separation and independence.
- An architecture in which components are grouped into layers, and
  - Components communicate only with other components in the layer immediately above and below their own layer.

# Layered Architecture

Name	Layered architecture
Description	Organizes the system into layers, with related functionality associated with each layer. A layer provides services to the layer above it, so the lowest level layers represent core services that are likely to be used throughout the system. See Figure 6.8.
Example	A layered model of a digital learning system to support learning of all subjects in schools (Figure 6.9).
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multilevel security.
Advantages	Allows replacement of entire layers as long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult, and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

# Generic Layered Architecture



User interface

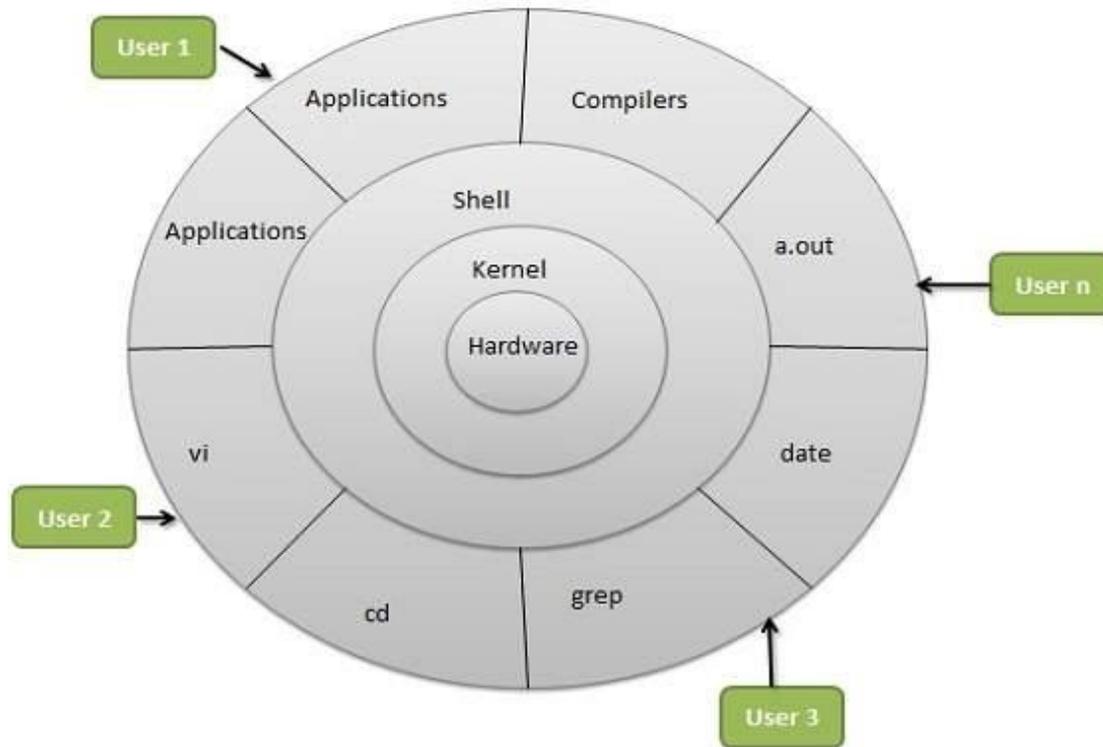
User interface management  
Authentication and authorization

Core business logic/application functionality  
System utilities

System support (OS, database, etc.)

# Layered Architecture

- The architecture of a Linux System consists of following layers:
  - **Hardware layer** (e.g. RAM/ CPU).
  - **Kernel** : it is the core component of OS
  - **Shell**: an interface to kernel, hiding complexity of kernel's functions.
  - **Utilities**: programs that provide the user most of the functionalities of OS.



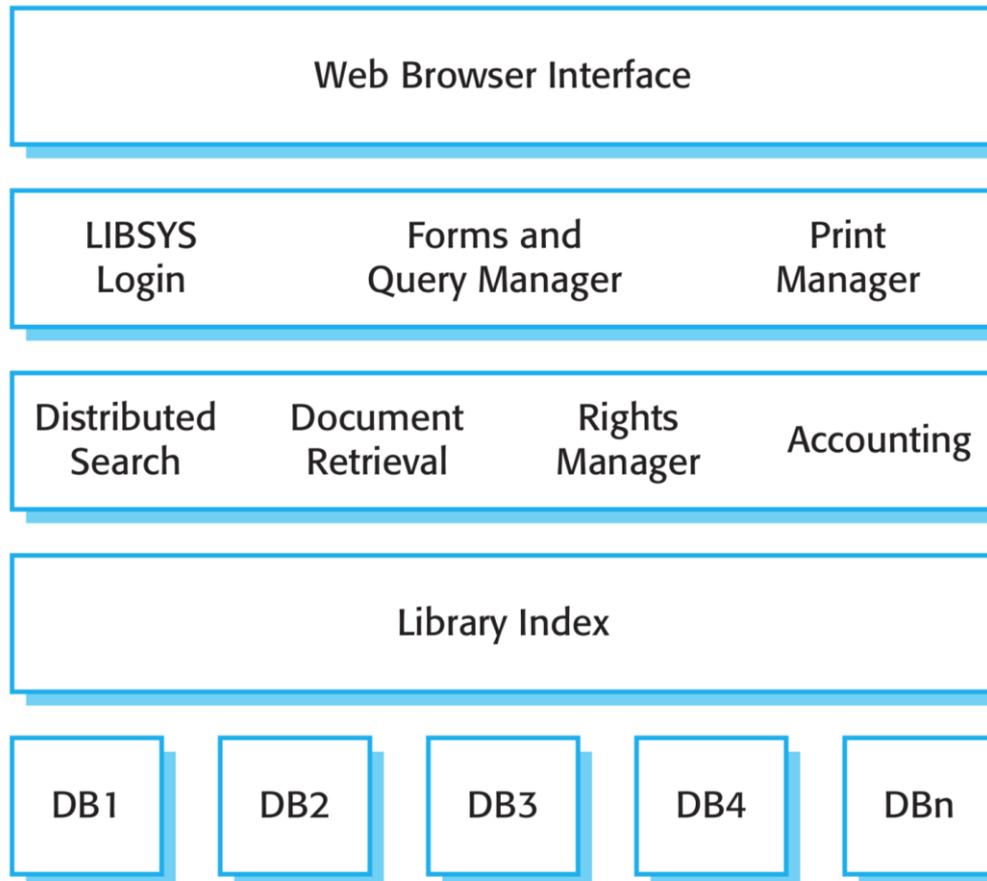
# Layered Architecture



- While layered architecture keeps the components themselves focused on specific tasks and facilitates the detection of problems;
  - It sometimes presents a **performance** problem in terms of the number of layers a message may have to travel through before being processed.

# Layered Architecture

- Library system (LIBSYS) as a layered architecture.



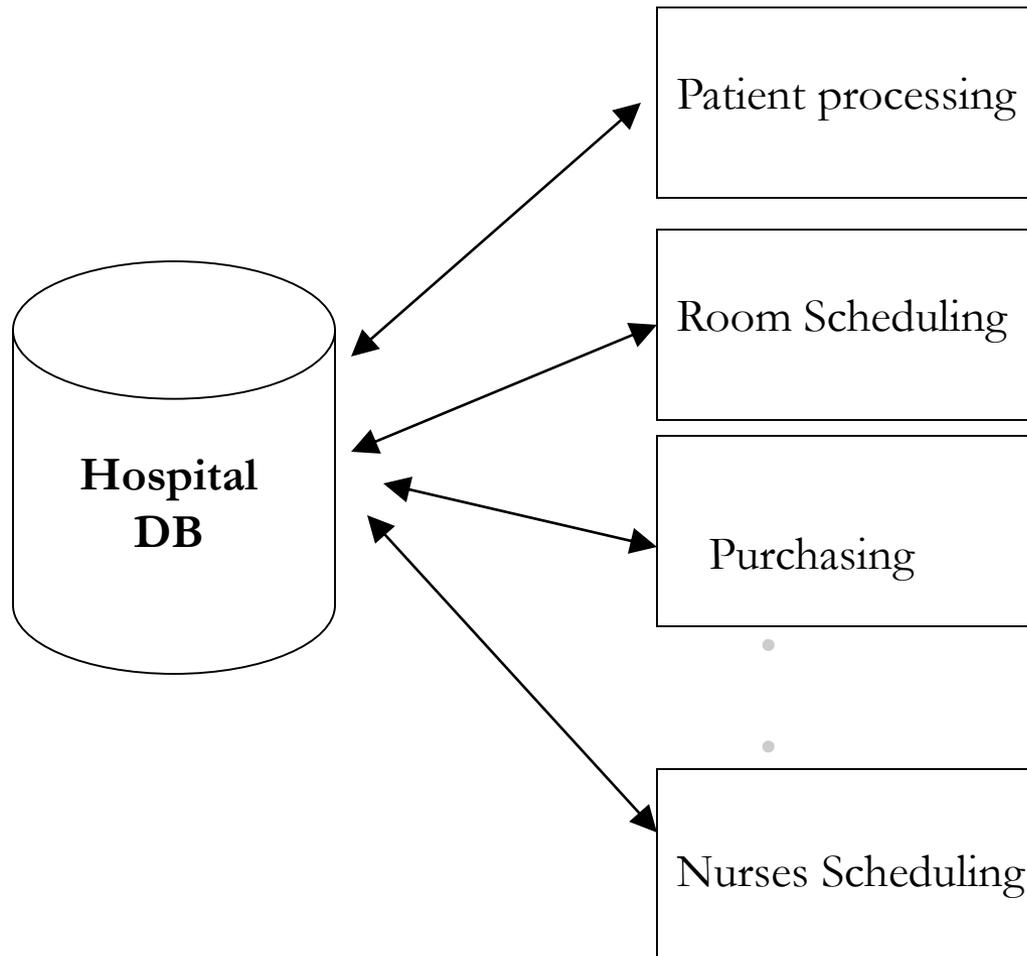
# Repository Architecture

- An architecture in which a central database and separate programs access the database.
- The programs communicate only through the database
  - Not directly among themselves.
- A big advantage – it introduces a layer of abstraction for the database.
  - Called a Database Management System (DBMS).

# Repository Architecture

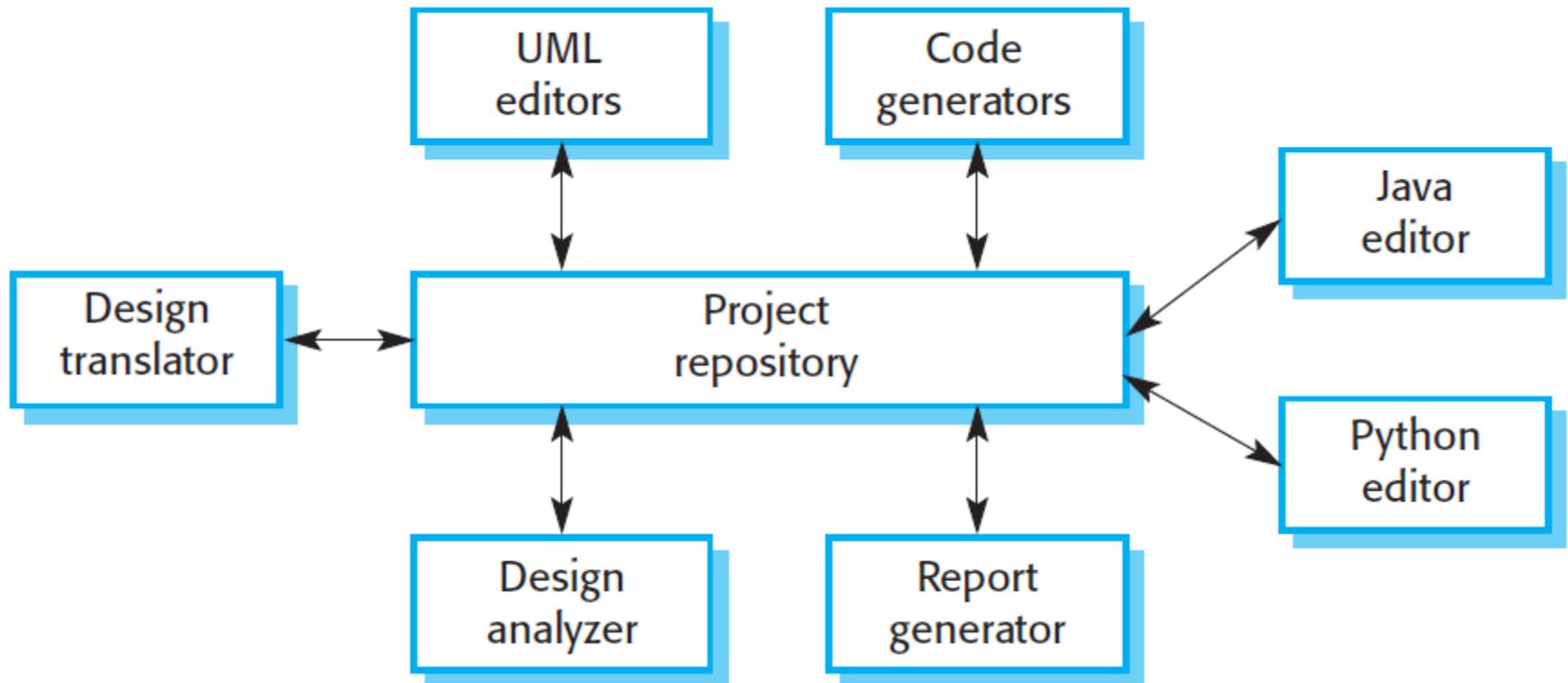
Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.11 is an example of an IDE where the components use a repository of system design information. Each software tool generates information, which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent; they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

# Repository Architecture



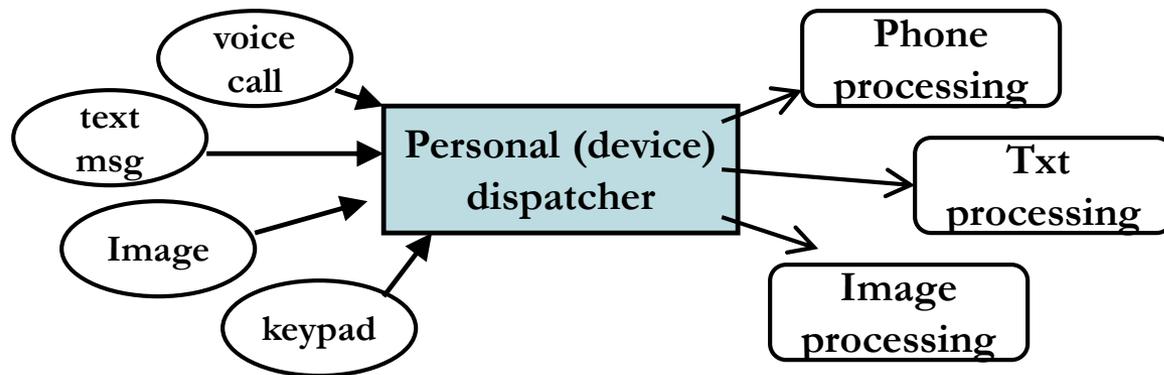
**Very popular  
within the  
business  
applications  
community**

# Repository Architecture for an IDE



# Event-Driven Architecture (Realtime)

- The high level design solution is based on an **event dispatcher** which manages events and the functionalities which depends on those events.



Problems that fit this architecture includes **real-time systems** such as: airplane control; medical equipment monitor; home monitor; embedded device controller; game; etc.

# Event-Driven Architecture Example

- Think of Java?
  - Java Swing API

```
public class FooPanel extends JPanel implements ActionListener {
    public FooPanel() {
        super();

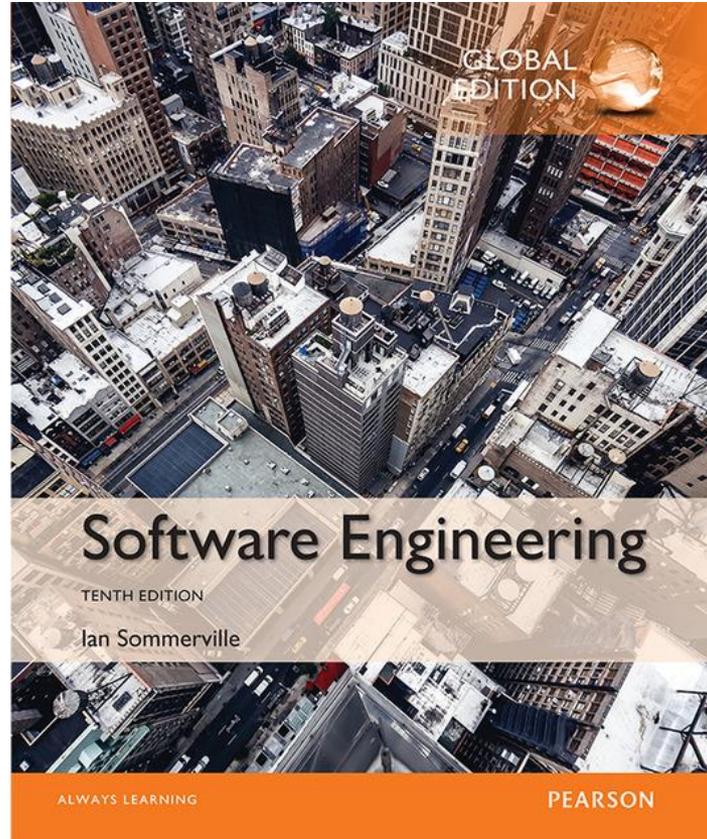
        JButton btn = new JButton("Click Me!");
        btn.addActionListener(this);

        this.add(btn);
    }

    @Override
    public void actionPerformed(ActionEvent ae) {
        System.out.println("Button has been clicked!");
    }
}
```

# Read

## Chapter 6

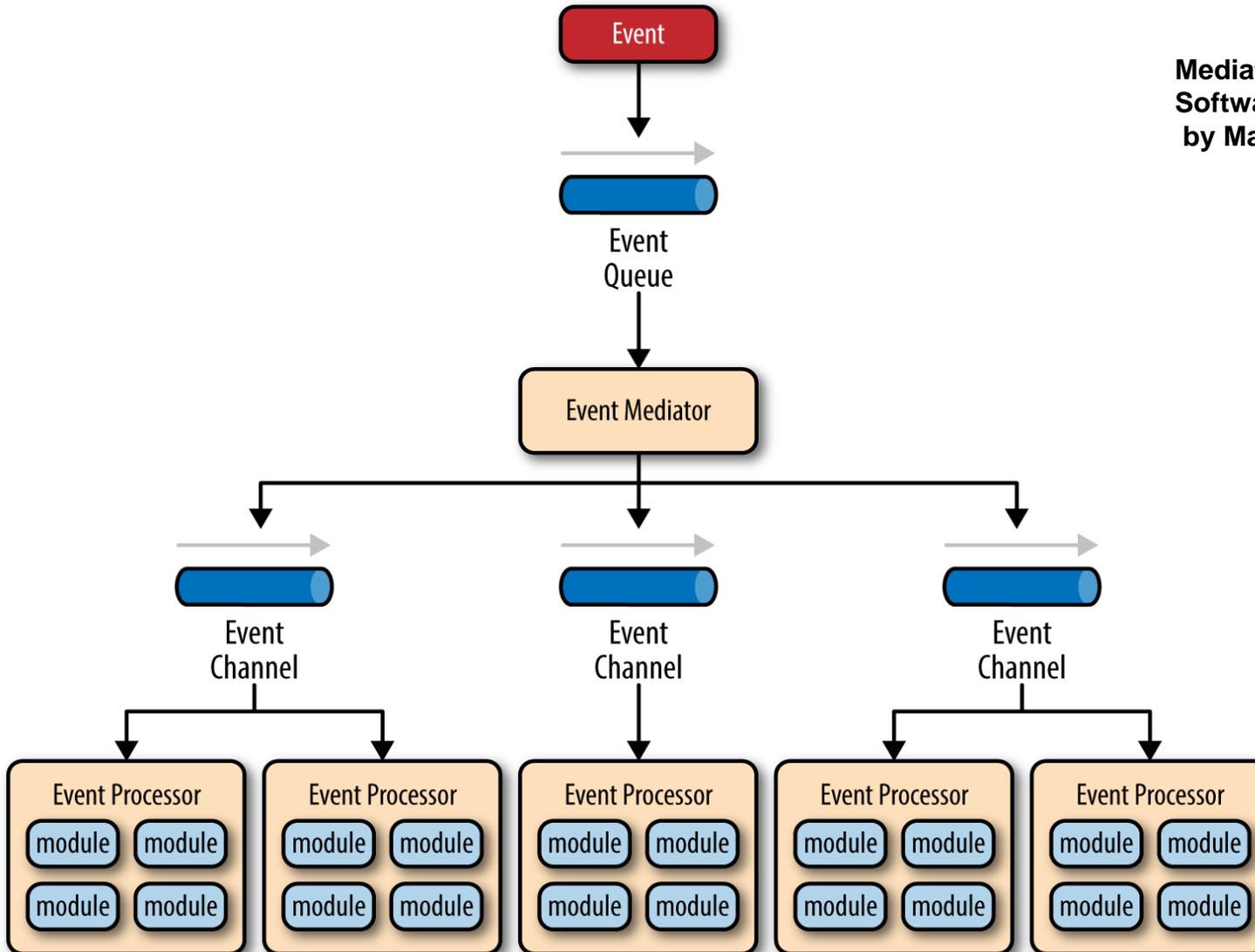


# References



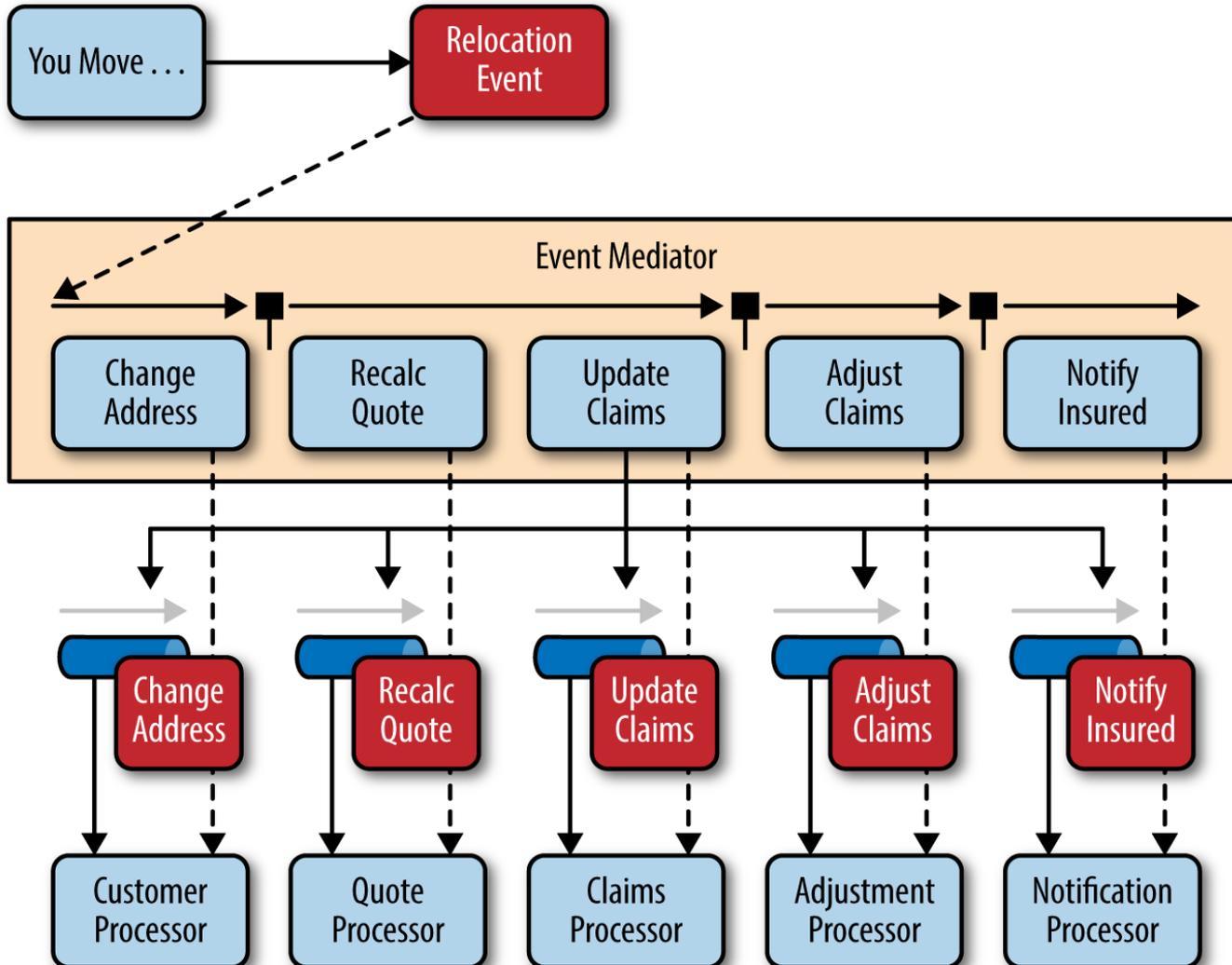
- Ian Sommerville, “Software Engineering”, 10<sup>th</sup> Edition, Addison-Wesley, 2015.
  - Timothy C. Lethbridge and Robert Laganière, “Object-Oriented Software Engineering: Practical Software Development using UML and Java”, 2<sup>nd</sup> Edition, McGraw Hill, 2001.
  - R. S. Pressman, Software Engineering: A Practitioner’s Approach, 10th Edition, McGraw-Hill, 2005.
- 

# Event-Driven Architecture (Realtime)



Mediator Topology:  
Software Architecture patterns  
by Mark Richards

# Event-Driven Architecture (Realtime)



Mediator Topology Example  
Software Architecture patterns  
by Mark Richards