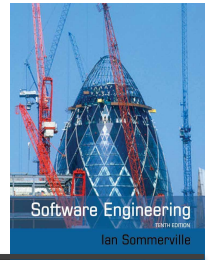


## UNIT-II

**System Modeling:** Context models, Interaction models, Structural models, Behavioural models, Model driven architecture.

**Architectural Design:** Design decisions, Architectural views, Architectural patterns and architectures

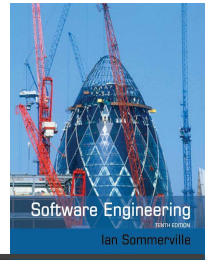
**Design and implementation:** Object oriented design using UML, Design patterns, Implementation issues, Open-source development



# Chapter 6 – Architectural Design

# Topics covered

---



- ✧ Architectural design decisions
- ✧ Architectural views
- ✧ Architectural patterns
- ✧ Application architectures

# Architectural design



- ✧ Architectural design is concerned with understanding how a software system should be **organized and designing the overall structure** of that system.
- ✧ Architectural design is the critical **link between design and requirements engineering**, as it identifies the **main structural components** in a system and the **relationships between them**.
- ✧ The **output** of the architectural design process is an **architectural model** that describes **how the system is organized as a set of communicating components**.

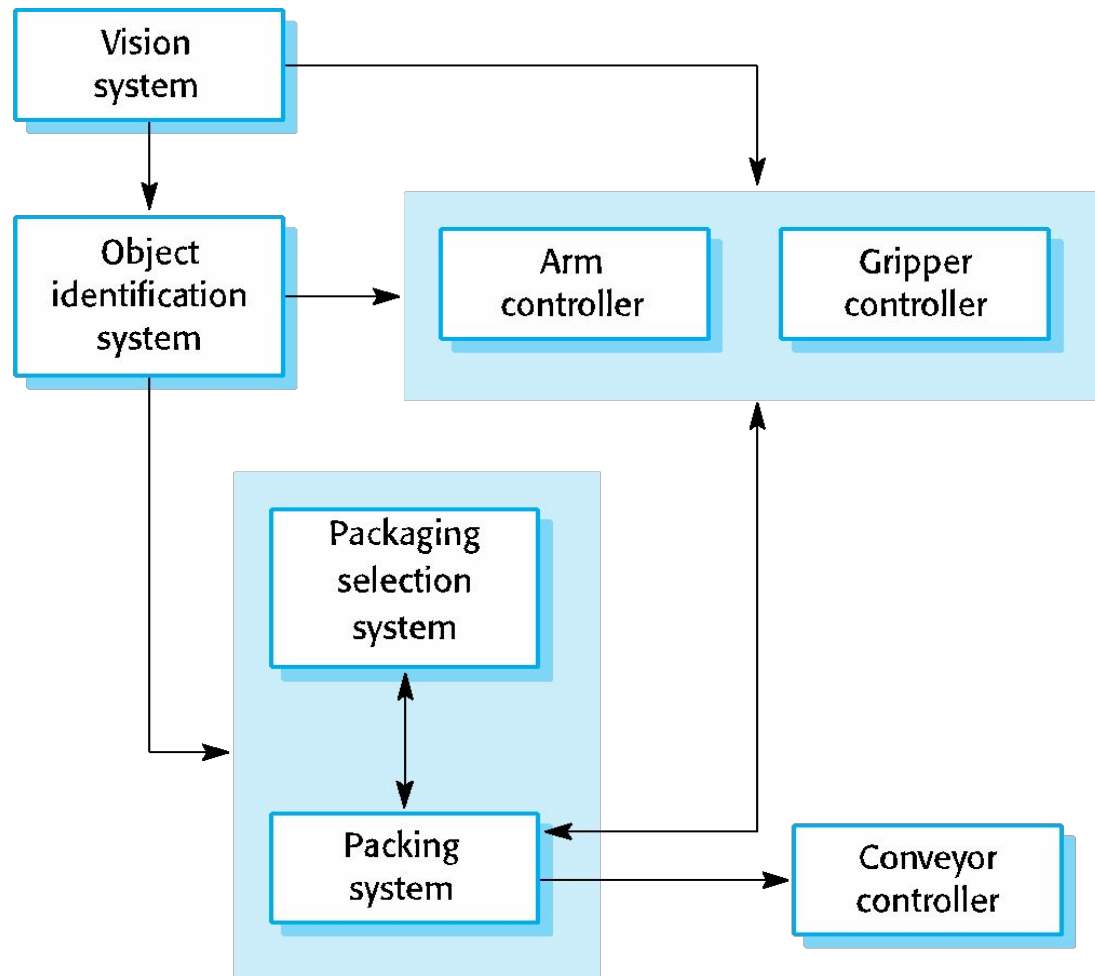
# Agility and architecture



- ✧ It is generally accepted that an early stage of agile processes is to design an overall systems architecture.
- ✧ Refactoring the system architecture is usually expensive because it affects so many components in the system

Software architecture is important because it affects the performance, robustness, distributability, and maintainability of a system (Bosch 2000). As Bosch explains, individual components implement the functional system requirements, but the dominant influence on the non-functional system characteristics is the system's architecture. Chen et al. (Chen, Ali Babar, and Nuseibeh 2013) confirmed this in a study of “architecturally significant requirements” where they found that non-functional requirements had the most significant effect on the system's architecture.

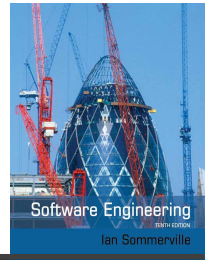
# The architecture of a packing robot control system



You can design software architectures at two levels of abstraction, which I call architecture in the small and architecture in the large:

## Architectural abstraction

---



- ✧ Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- ✧ Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

# 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
  - Product-line architectures may be developed.



# Architectural representations



- ✧ Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures.
- ✧ But these have been criticised because they lack semantics, do not show the types of relationships between entities nor the visible properties of entities in the architecture.
- ✧ Depends on the use of architectural models. The requirements for model semantics depends on how the models are used.

unclear  
conveying of  
components

# Box and line diagrams

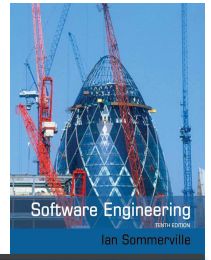
---



- ✧ 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.

The apparent contradictions between architectural theory and industrial practice arise because there are two ways in which an architectural model of a program is used:

## Use of architectural models



- ✧ As a way of facilitating discussion about the system design
  - A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.
- ✧ As a way of documenting an architecture that has been designed
  - The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.

# Architectural design decisions

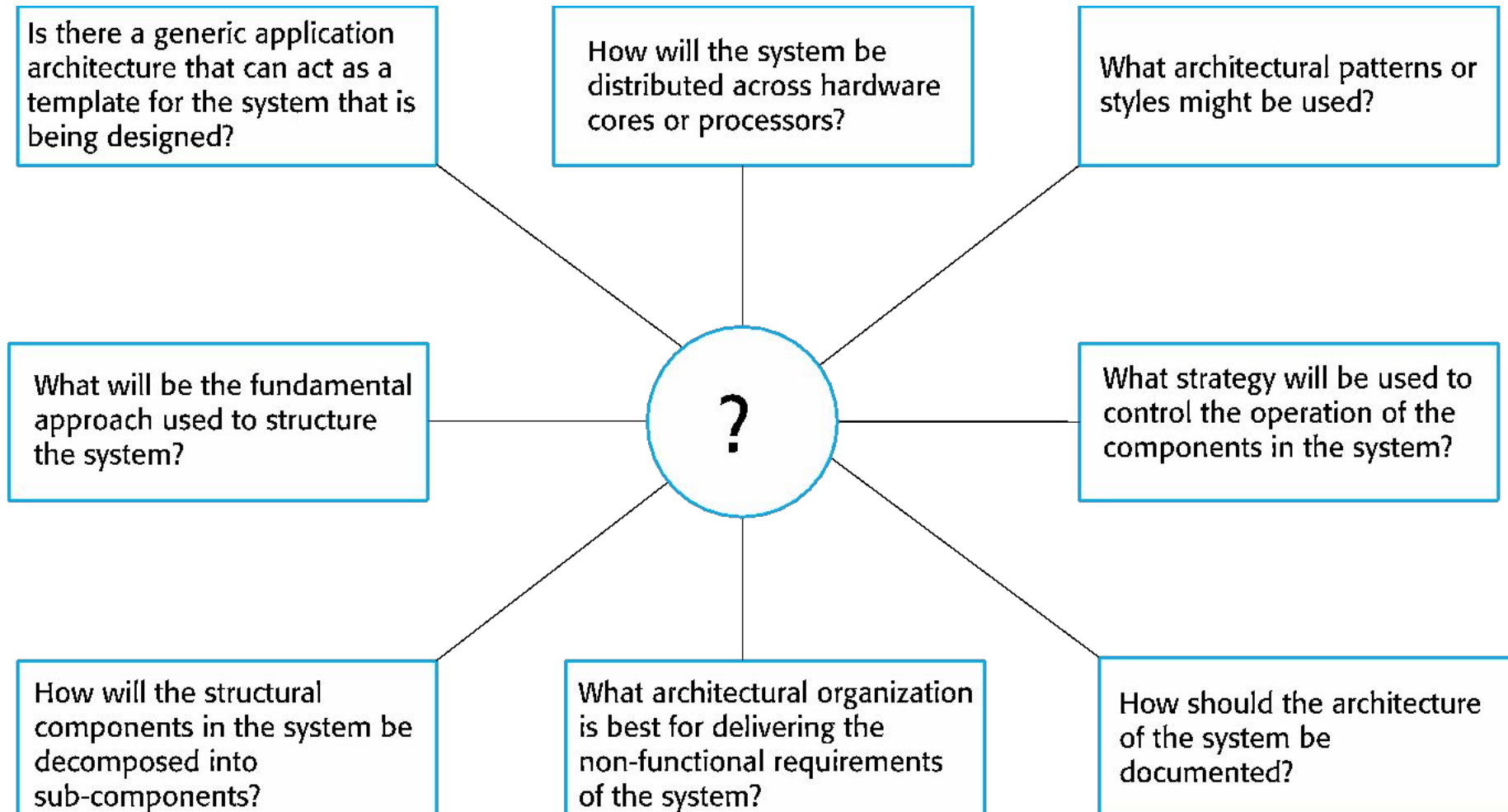
# Architectural design decisions

---



- ✧ Architectural design is a creative process so the process differs depending on the type of system being developed.
- ✧ However, a number of common decisions span all design processes and these decisions affect the non-functional characteristics of the system.

# Architectural design decisions

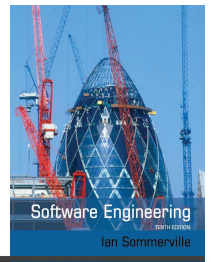


# Architecture reuse



- ✧ Systems in the same domain often have similar architectures that reflect domain concepts.
- ✧ Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
- ✧ The architecture of a system may be designed around one of more architectural patterns or 'styles'.
  - These capture the essence of an architecture and can be instantiated in different ways.

# Architecture and system characteristics



## ✧ Performance

- Localise critical operations and minimise communications. Use large rather than fine-grain components.

## ✧ Security

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

## ✧ Safety

- Localise safety-critical features in a small number of sub-systems.

## ✧ Availability

- Include redundant components and mechanisms for fault tolerance.

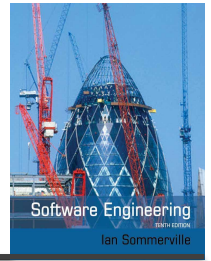
## ✧ Maintainability

- Use fine-grain, replaceable components.



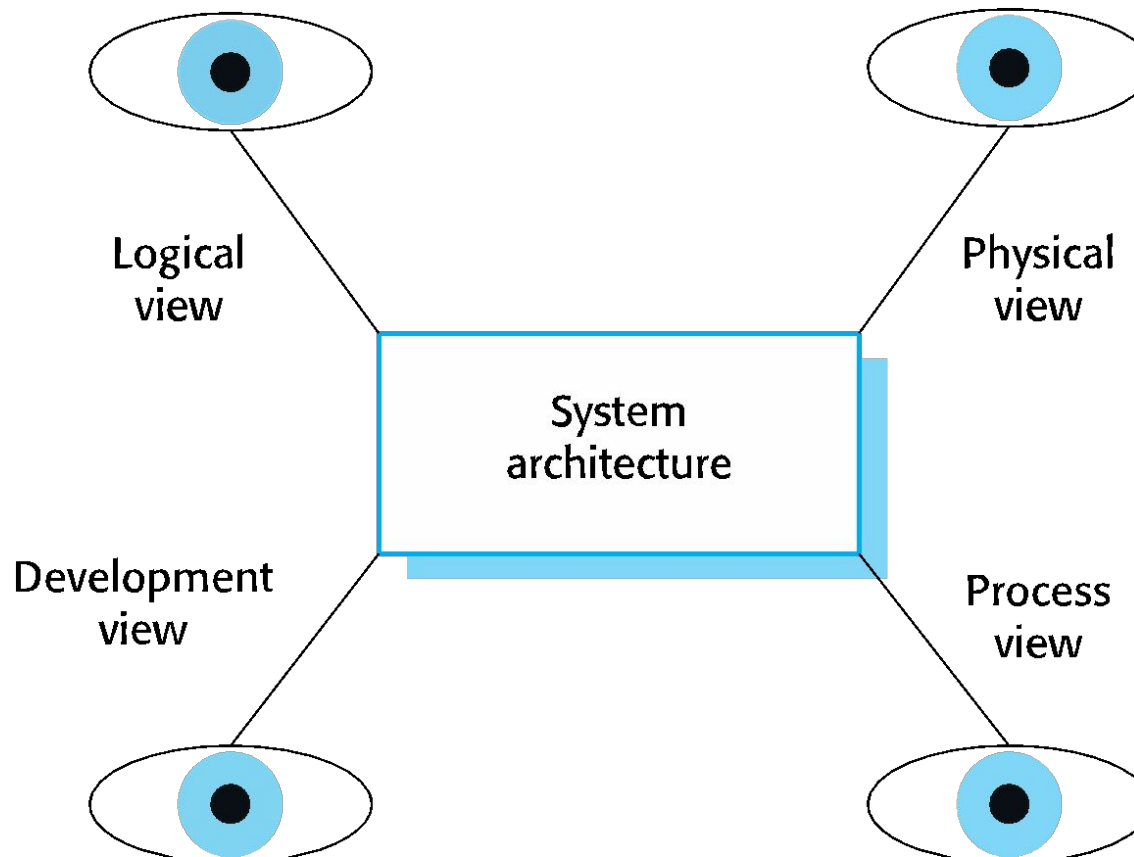
# Architectural views

# Architectural views



- ✧ What views or perspectives are useful when designing and documenting a system's architecture?
- ✧ What notations should be used for describing architectural models?
- ✧ 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 or the 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

---



- ✧ A logical view, which shows the key abstractions in the system as objects or object classes.
- ✧ A process view, which shows how, at run-time, the system is composed of interacting processes.
- ✧ A development view, which shows how the software is decomposed for development.
- ✧ A physical view, which shows the system hardware and how software components are distributed across the processors in the system.
- ✧ Related using use cases or scenarios (+1)

# Representing architectural views



- ✧ Some people argue that the Unified Modeling Language (UML) is an appropriate notation for describing and documenting system architectures
- ✧ I disagree with this as I do not think that the UML includes abstractions appropriate for high-level system description.
- ✧ Architectural description languages (ADLs) have been developed but are not widely used

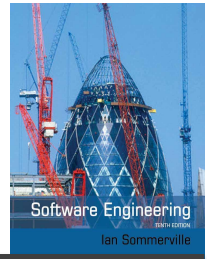
# Architectural patterns

# Architectural patterns



- ✧ Patterns are a means of representing, sharing and reusing knowledge.
- ✧ An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- ✧ Patterns should include information about when they are and when they are not useful.
- ✧ Patterns may be represented using tabular and graphical descriptions.

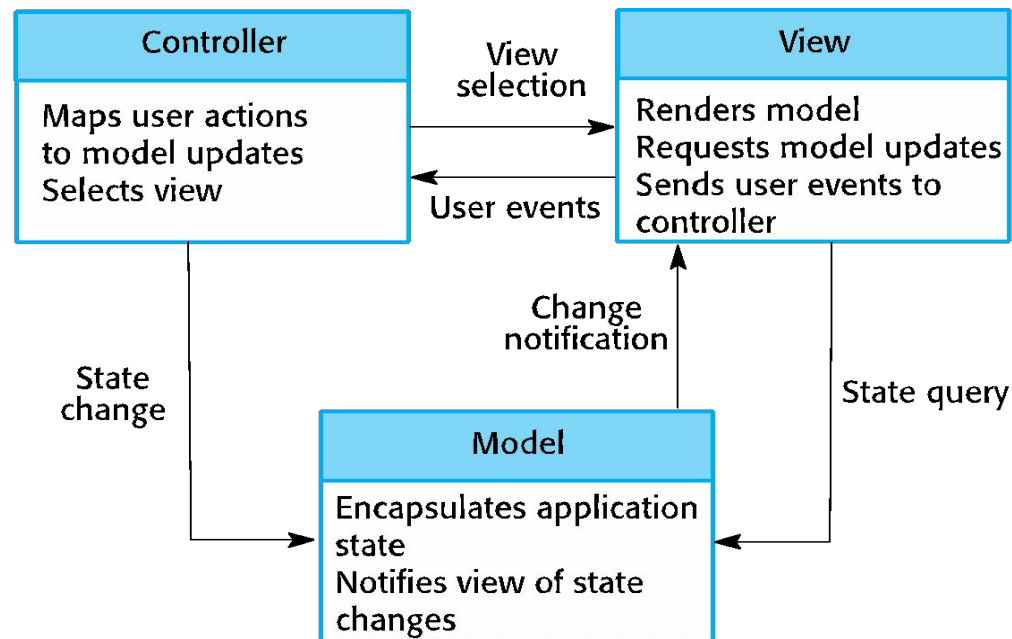
# The Model-View-Controller (MVC) pattern



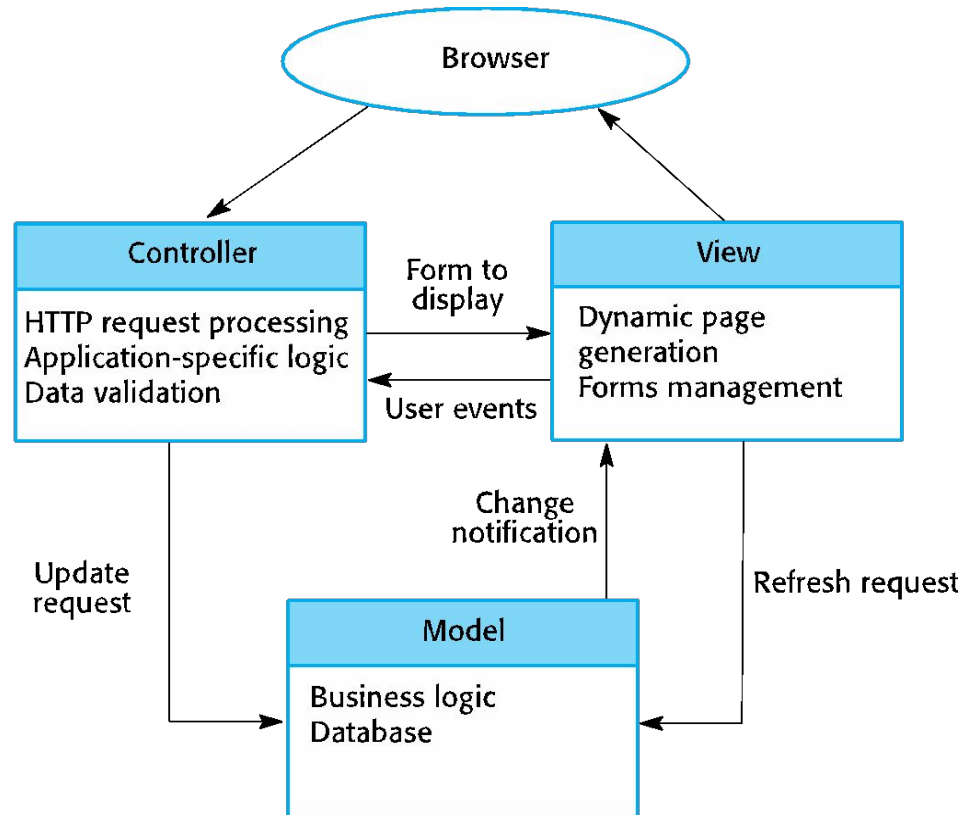
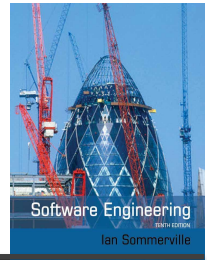
| Name                 | MVC (Model-View-Controller)   |
|----------------------|---|
| <b>Description</b>   | Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3. |
| <b>Example</b>       | Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.  |
| <b>When used</b>     | Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.  |
| <b>Advantages</b>    | Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.  |
| <b>Disadvantages</b> | Can involve additional code and code complexity when the data model and interactions are simple.  |



# The organization of the Model-View-Controller



# Web application architecture using the MVC pattern



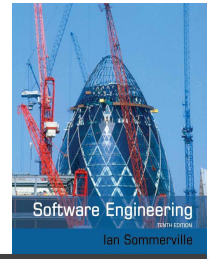
# Layered architecture

---



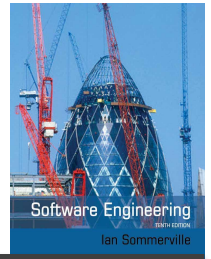
- ✧ Used to model the interfacing of sub-systems.
- ✧ Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- ✧ Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- ✧ However, often artificial to structure systems in this way.

# The Layered architecture pattern



| 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 6.6.  |
| Example       | A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.  |
| 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 multi-level security.   |
| Advantages    | Allows replacement of entire layers so 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. |

# A generic layered architecture



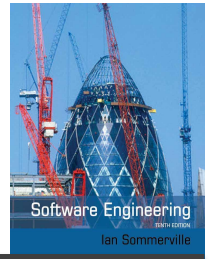
User interface

User interface management  
Authentication and authorization

Core business logic/application functionality  
System utilities

System support (OS, database etc.)

# The architecture of the iLearn system



Browser-based user interface

iLearn app

## Configuration services

Group  
management

Application  
management

Identity  
management

## Application services

Email   Messaging   Video conferencing   Newspaper archive  
Word processing   Simulation   Video storage   Resource finder  
Spreadsheet   Virtual learning environment   History archive

## Utility services

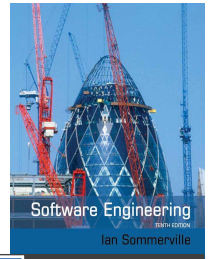
Authentication   Logging and monitoring   Interfacing  
User storage   Application storage   Search

# Repository architecture



- ✧ Sub-systems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- ✧ When large amounts of data are to be shared, the repository model of sharing is most commonly used as this is an efficient data sharing mechanism.

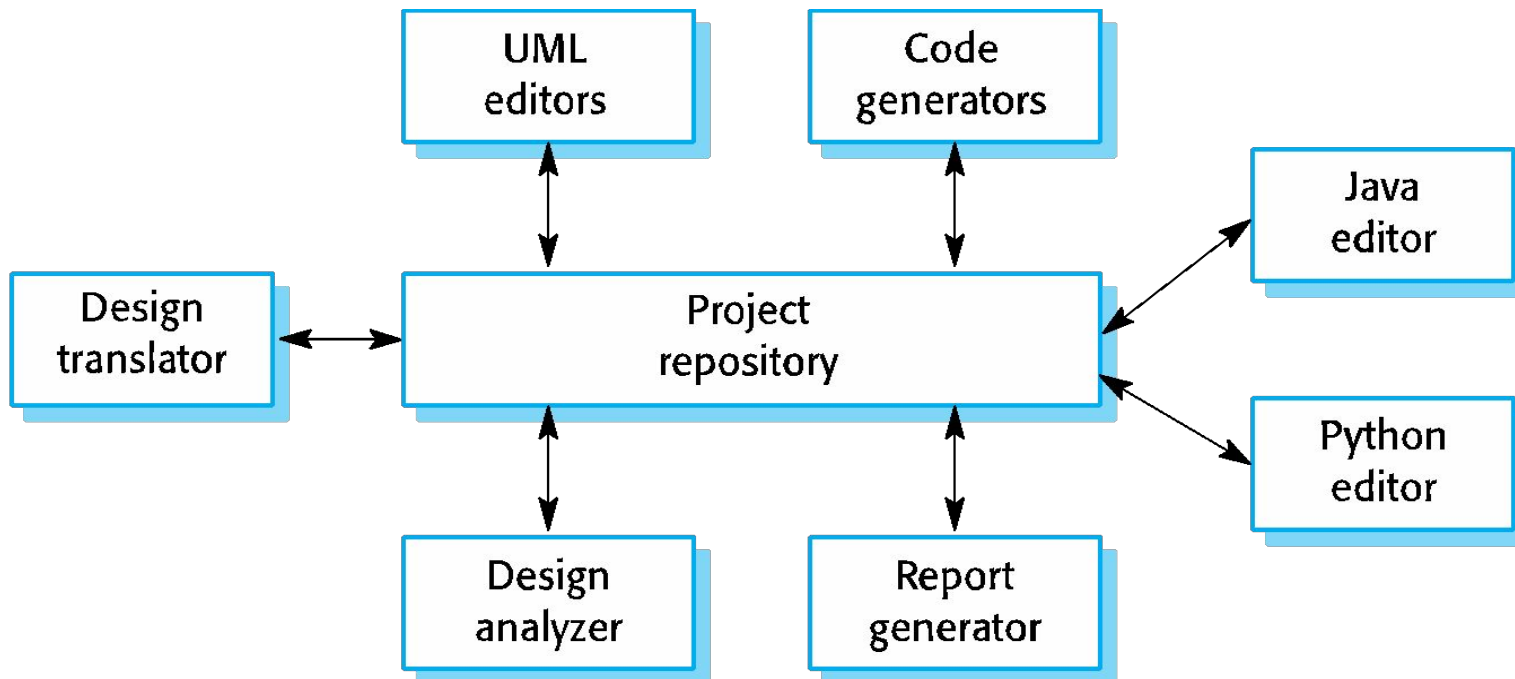
# The Repository pattern



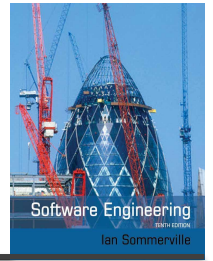
| Name          | Repository  |
|---------------|---|
| Description   | All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.  |
| Example       | Figure 6.9 is an example of an IDE where 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.               |



# A repository architecture for an IDE

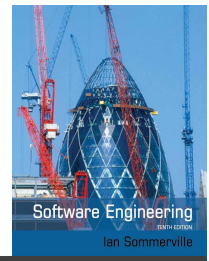


# Client-server architecture



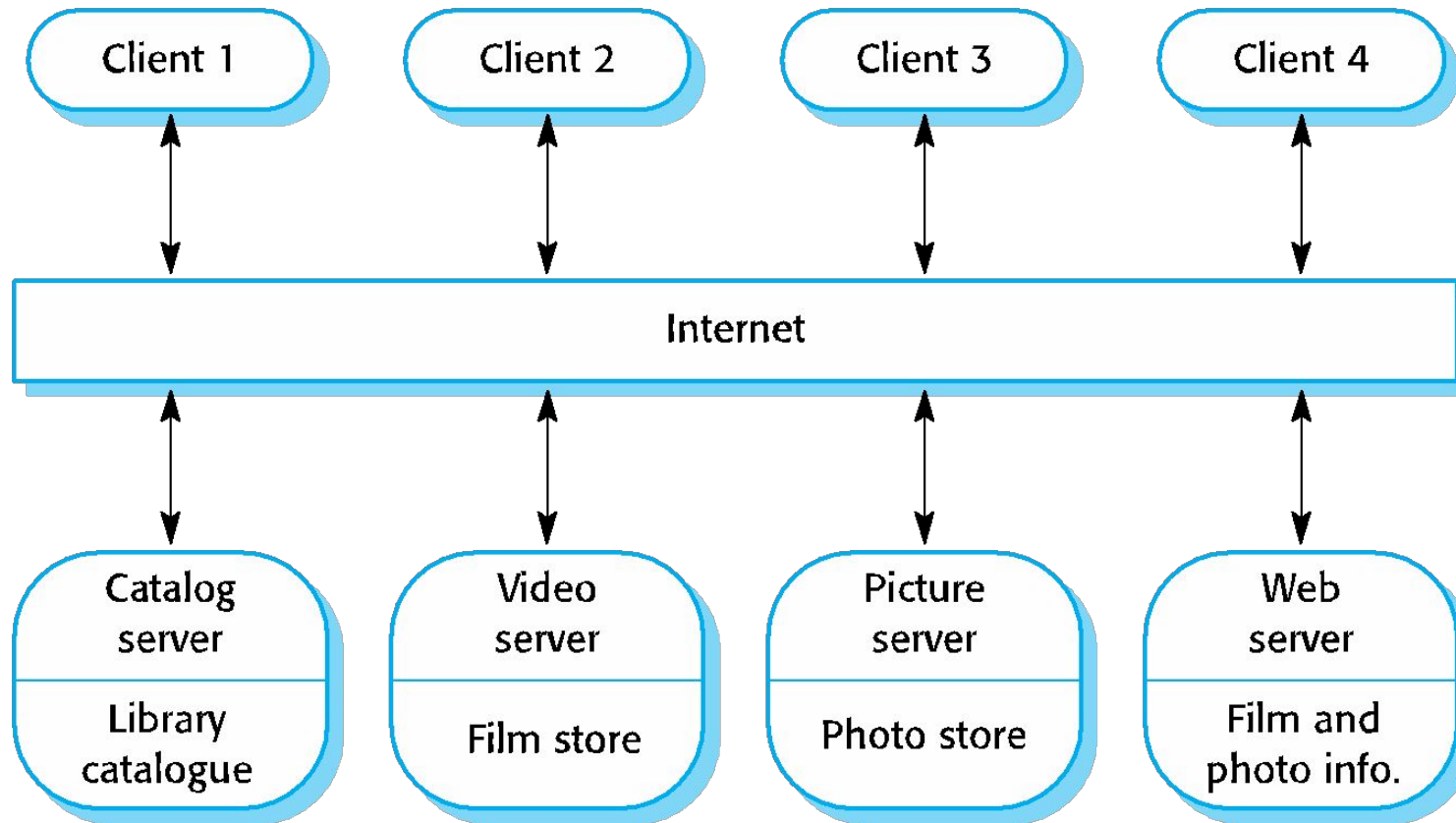
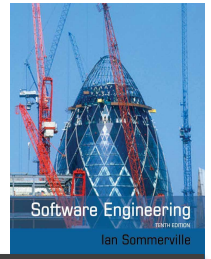
- ✧ Distributed system model which shows how data and processing is distributed across a range of components.
  - Can be implemented on a single computer.
- ✧ Set of stand-alone servers which provide specific services such as printing, data management, etc.
- ✧ Set of clients which call on these services.
- ✧ Network which allows clients to access servers.

# The Client–server pattern



| Name          | Client-server  |
|---------------|--|
| Description   | In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.   |
| Example       | Figure 6.11 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 so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations. |

# A client-server architecture for a film library

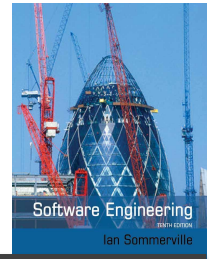


# Pipe and filter architecture



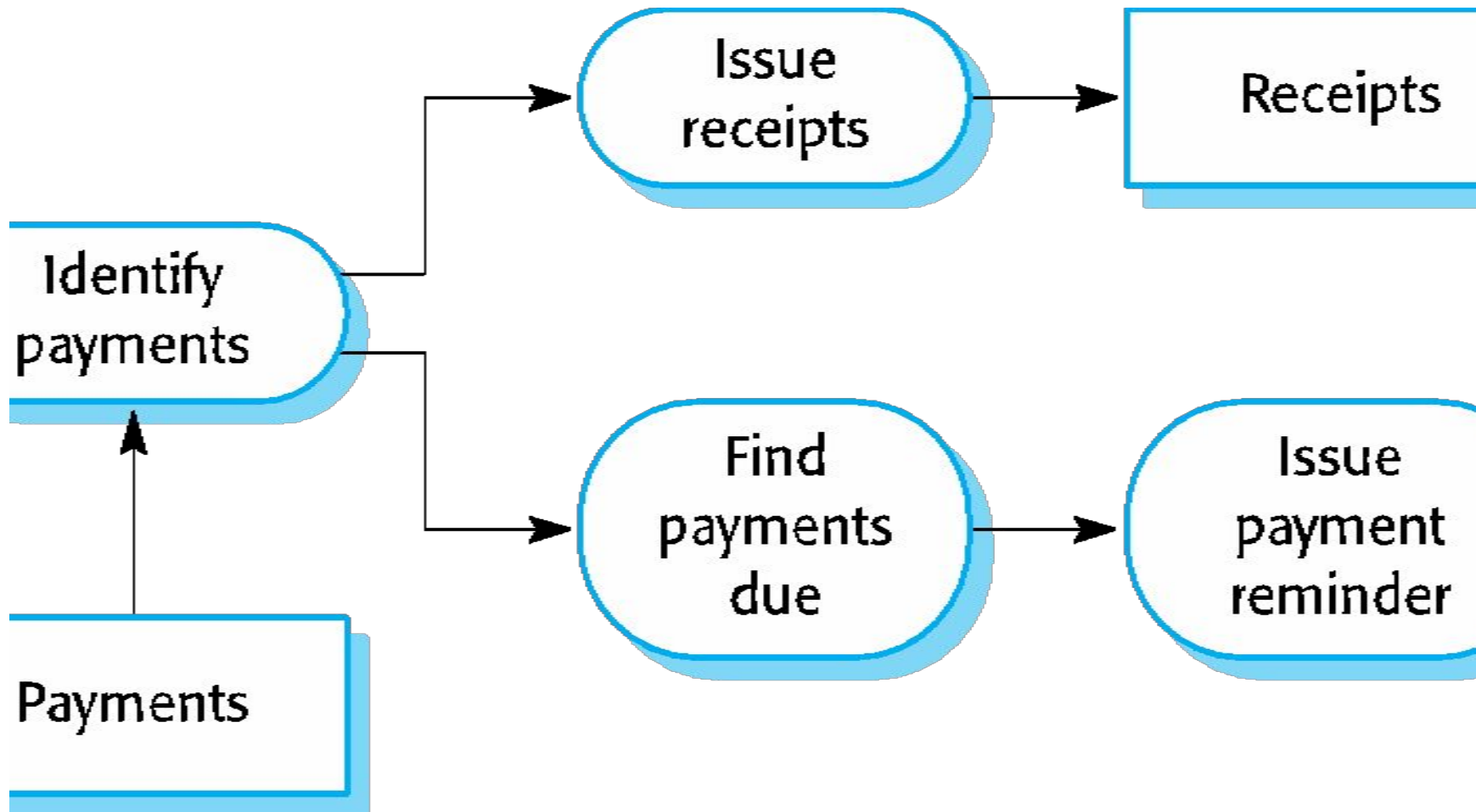
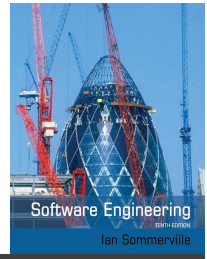
- ✧ Functional transformations process their inputs to produce outputs.
- ✧ May be referred to as a pipe and filter model (as in UNIX shell).
- ✧ Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- ✧ Not really suitable for interactive systems.

# The pipe and filter pattern



| Name          | Pipe and filter  |
|---------------|--|
| Description   | The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.  |
| Example       | Figure 6.13 is an example of a pipe and filter system used for processing invoices.  |
| When used     | Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.   |
| Advantages    | Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.   |
| Disadvantages | The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures. |

# An example of the pipe and filter architecture used in a payments system



# Application architectures



# Application architectures

---



- ✧ Application systems are designed to meet an organisational need.
- ✧ As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements.
- ✧ A generic application architecture is an architecture for a type of software system that may be configured and adapted to create a system that meets specific requirements.

# Use of application architectures

---



- ✧ As a starting point for architectural design.
- ✧ As a design checklist.
- ✧ As a way of organising the work of the development team.
- ✧ As a means of assessing components for reuse.
- ✧ As a vocabulary for talking about application types.



# Examples of application types

---

## ✧ Data processing applications

- Data driven applications that process data in batches without explicit user intervention during the processing.

## ✧ Transaction processing applications

- Data-centred applications that process user requests and update information in a system database.

## ✧ Event processing systems

- Applications where system actions depend on interpreting events from the system's environment.

## ✧ Language processing systems

- Applications where the users' intentions are specified in a formal language that is processed and interpreted by the system.

# Application type examples

---



- ✧ Two very widely used generic application architectures are transaction processing systems and language processing systems.
- ✧ Transaction processing systems
  - E-commerce systems;
  - Reservation systems.
- ✧ Language processing systems
  - Compilers;
  - Command interpreters.

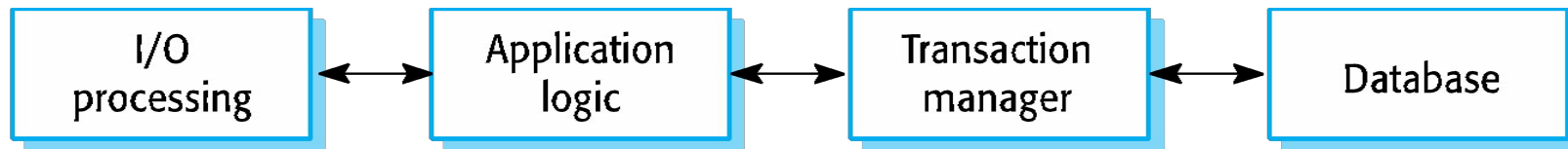
# Transaction processing systems

---

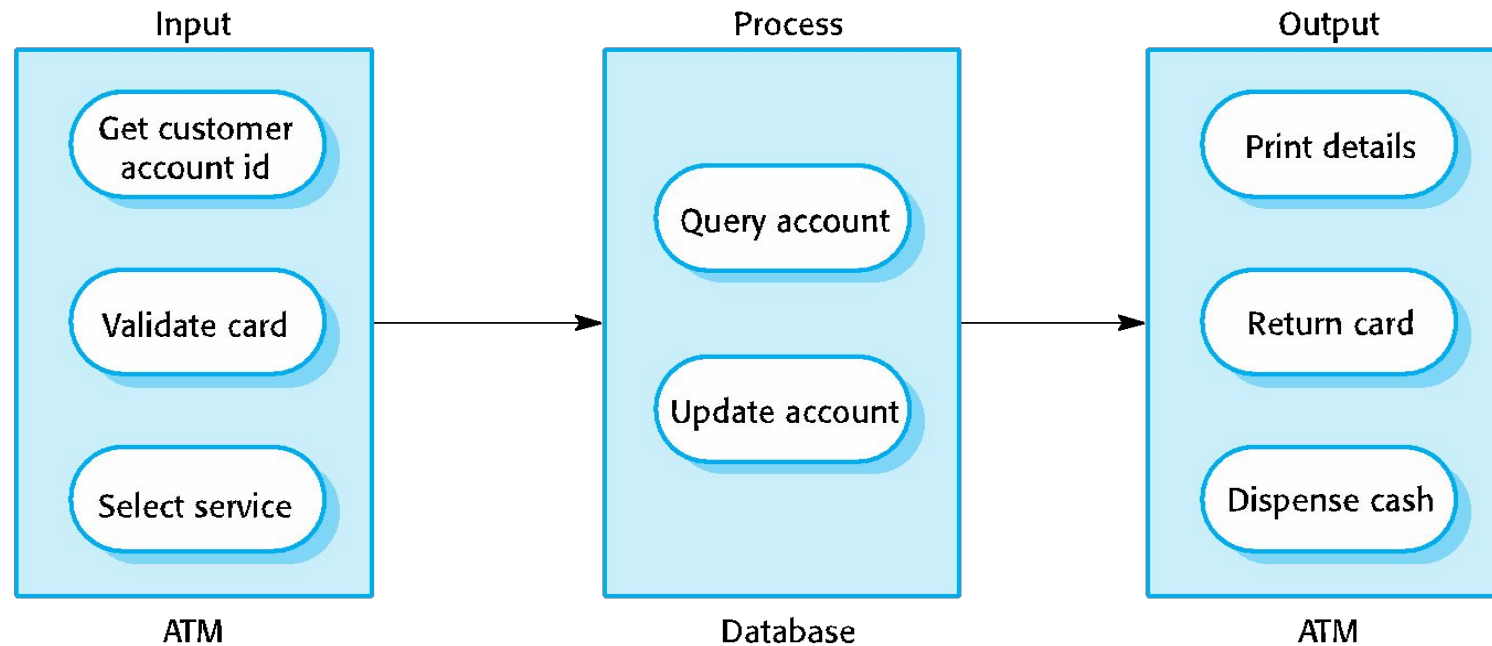


- ✧ Process user requests for information from a database or requests to update the database.
- ✧ From a user perspective a transaction is:
  - Any coherent sequence of operations that satisfies a goal;
  - For example - find the times of flights from London to Paris.
- ✧ Users make asynchronous requests for service which are then processed by a transaction manager.

# The structure of transaction processing applications



# The software architecture of an ATM system



# Information systems architecture

---



- ✧ Information systems have a generic architecture that can be organised as a layered architecture.
- ✧ These are transaction-based systems as interaction with these systems generally involves database transactions.
- ✧ Layers include:
  - The user interface
  - User communications
  - Information retrieval
  - System database



# Layered information system architecture



User interface

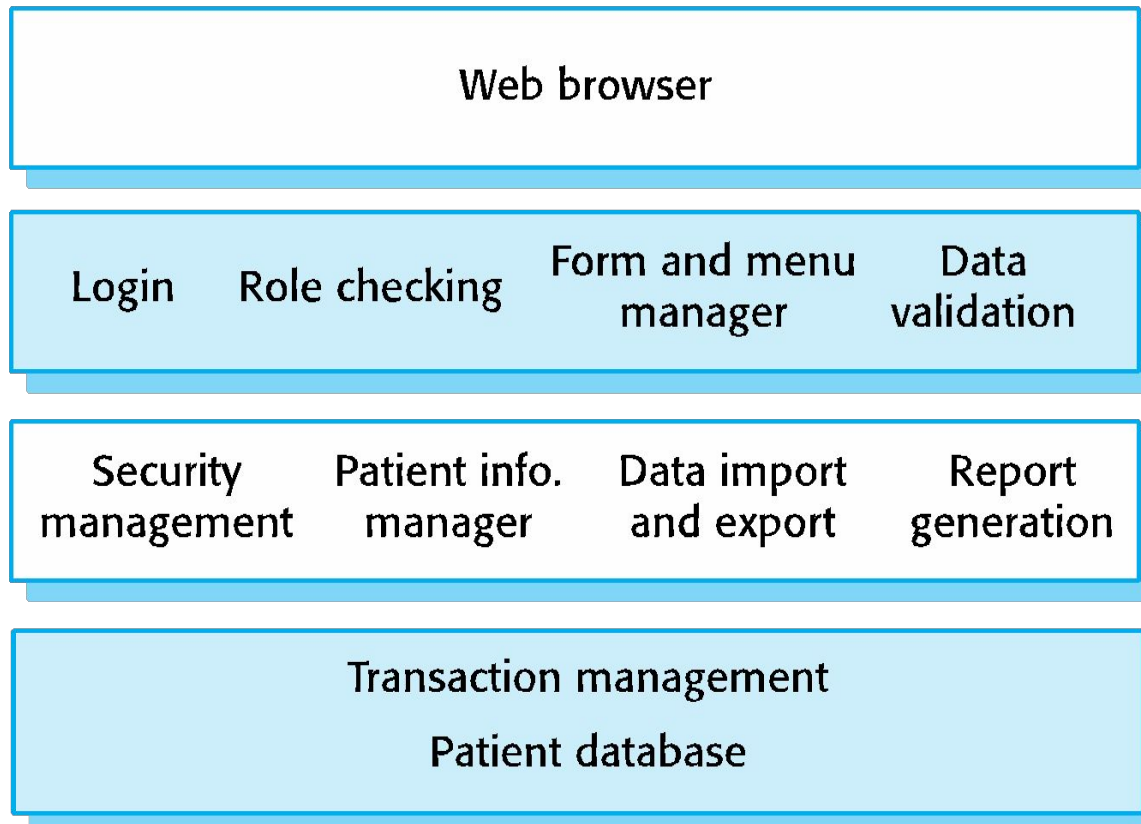
User communications

Authentication and  
authorization

Information retrieval and modification

Transaction management  
Database

# The architecture of the Mentcare system



# Web-based information systems



- ✧ Information and resource management systems are now usually web-based systems where the user interfaces are implemented using a web browser.
- ✧ For example, e-commerce systems are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery of these goods or services to the customer.
- ✧ In an e-commerce system, the application-specific layer includes additional functionality supporting a 'shopping cart' in which users can place a number of items in separate transactions, then pay for them all together in a single transaction.

# Server implementation



- ✧ These systems are often implemented as multi-tier client server/architectures (discussed in Chapter 17)
  - The web server is responsible for all user communications, with the user interface implemented using a web browser;
  - The application server is responsible for implementing application-specific logic as well as information storage and retrieval requests;
  - The database server moves information to and from the database and handles transaction management.

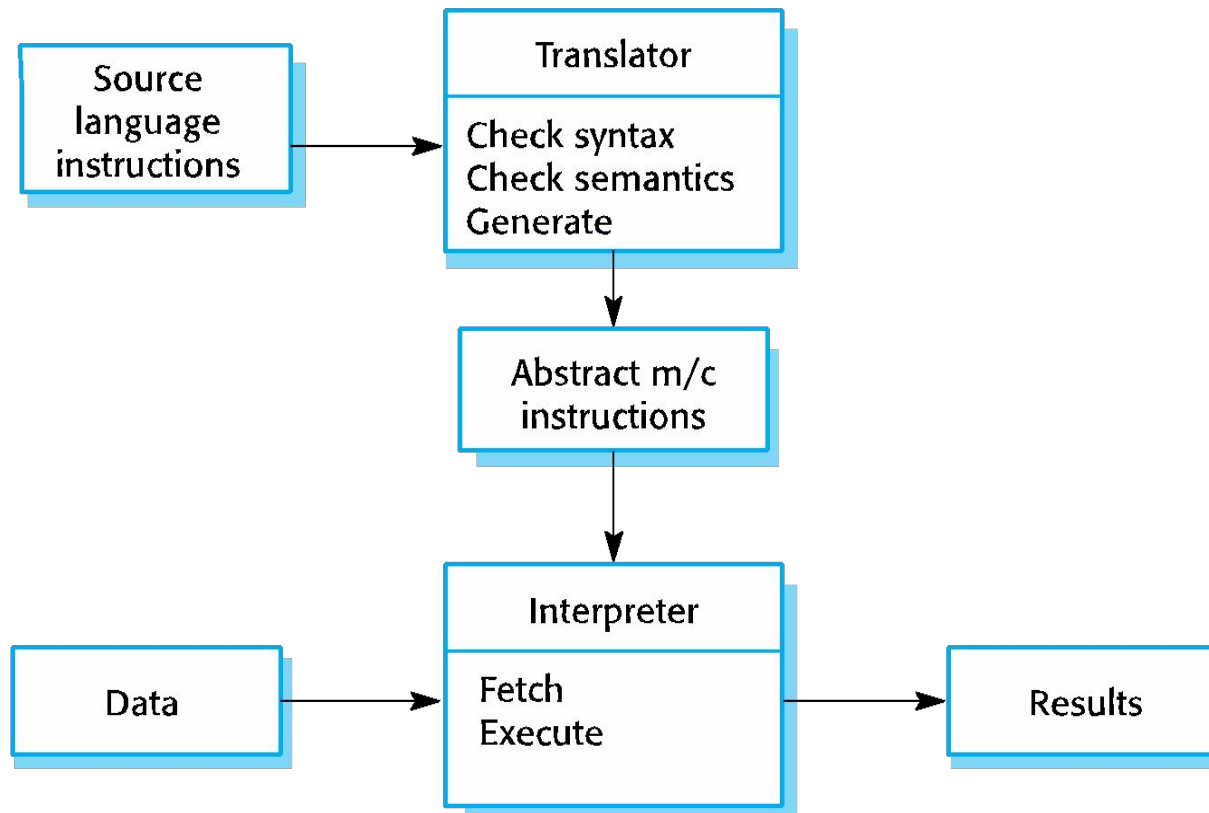
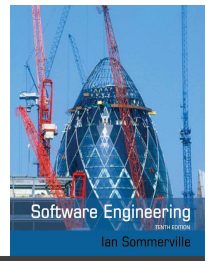
# Language processing systems

---



- ✧ Accept a natural or artificial language as input and generate some other representation of that language.
- ✧ May include an interpreter to act on the instructions in the language that is being processed.
- ✧ Used in situations where the easiest way to solve a problem is to describe an algorithm or describe the system data
  - Meta-case tools process tool descriptions, method rules, etc and generate tools.

# The architecture of a language processing system



# Compiler components



- ✧ A lexical analyzer, which takes input language tokens and converts them to an internal form.
- ✧ A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
- ✧ A syntax analyzer, which checks the syntax of the language being translated.
- ✧ A syntax tree, which is an internal structure representing the program being compiled.

# Compiler components

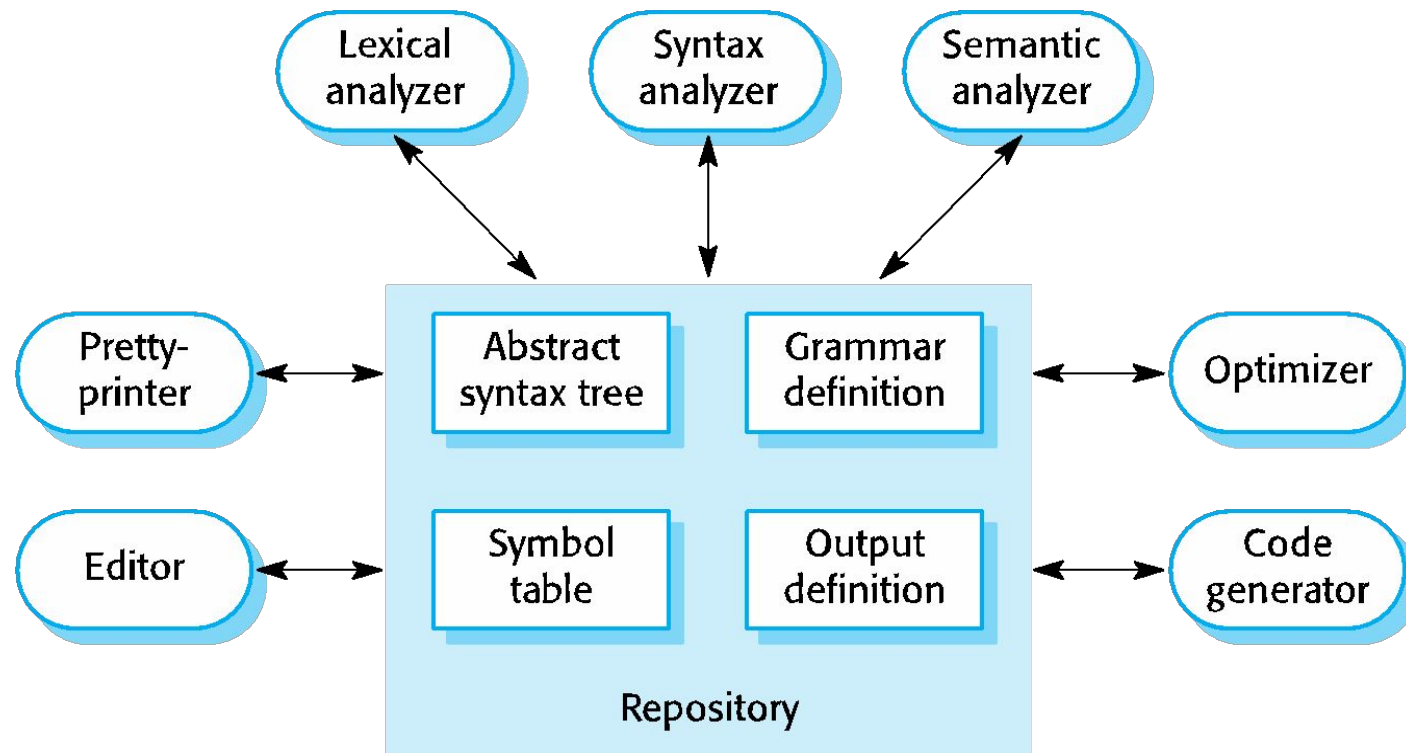
---



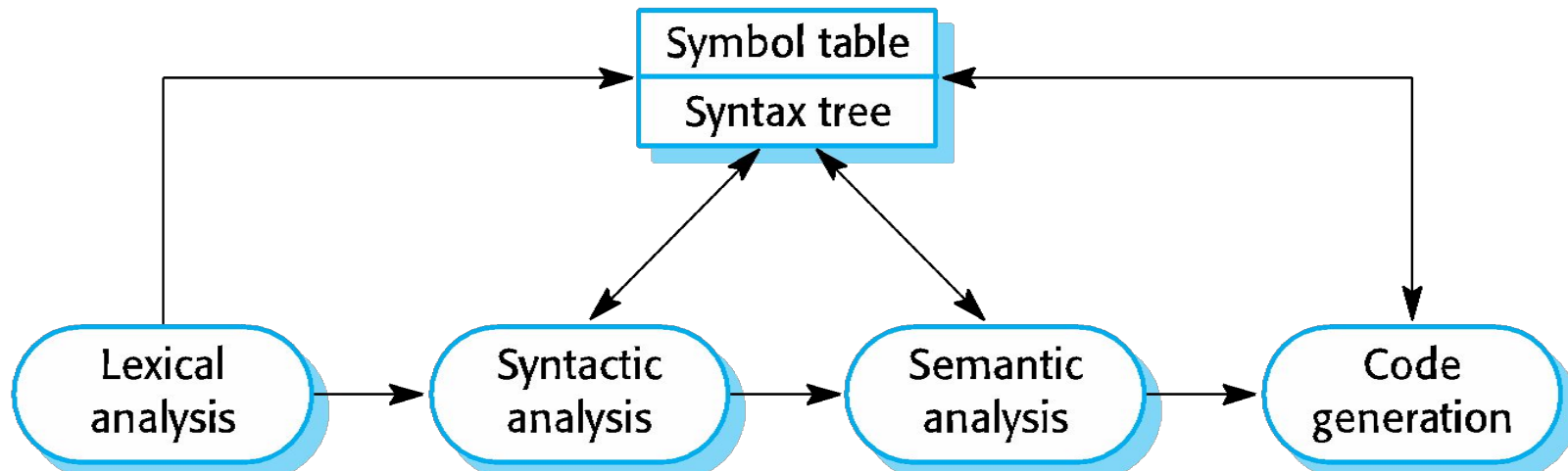
- ✧ A semantic analyzer that uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
- ✧ A code generator that 'walks' the syntax tree and generates abstract machine code.



# A repository architecture for a language processing system



# A pipe and filter compiler architecture



# Key points



- ✧ A software architecture is a description of how a software system is organized.
- ✧ Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used.
- ✧ Architectures may be documented from several different perspectives or views such as a conceptual view, a logical view, a process view, and a development view.
- ✧ Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used and describe its advantages and disadvantages.

# Key points



- ✧ Models of application systems architectures help us understand and compare applications, validate application system designs and assess large-scale components for reuse.
- ✧ Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified by a number of users.
- ✧ Language processing systems are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.