Architectural Design

Source

- Software Engineering 9th / 10th Edition, Ian Sommerville

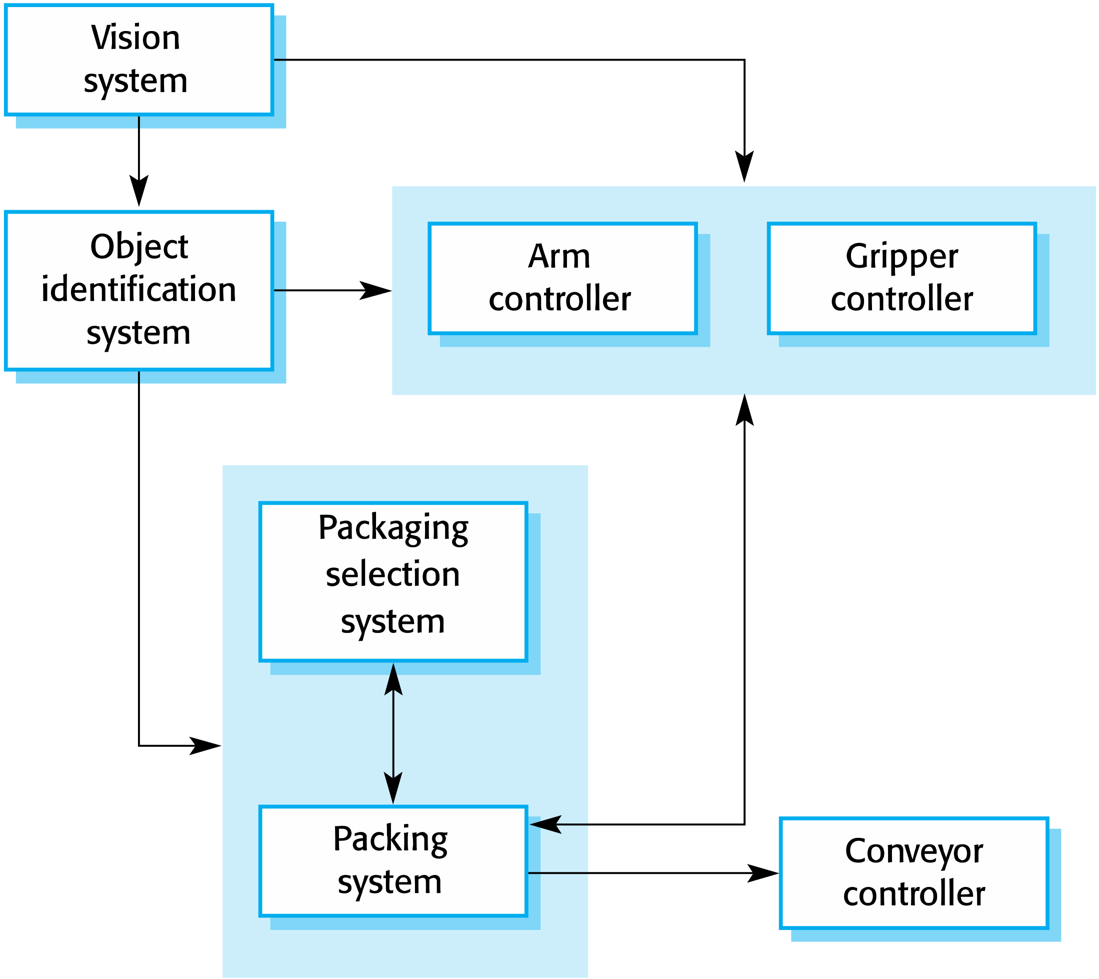
Topics covered

- Architectural design decisions
- Architectural views
- Architectural patterns

Architectural design

- 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.
- Architectural design is concerned with understanding how a software system should be **organized** and designing the **overall structure** of that system.
- The output of the architectural design process is an **architectural model** that describes how the system is organized as a set of **communicating components**.

The architecture of a packing robot control system



Architectural representations

- Simple, informal **block diagrams** showing **entities** and **relationships** are the most frequently used method for documenting software architectures.

Use of architectural models

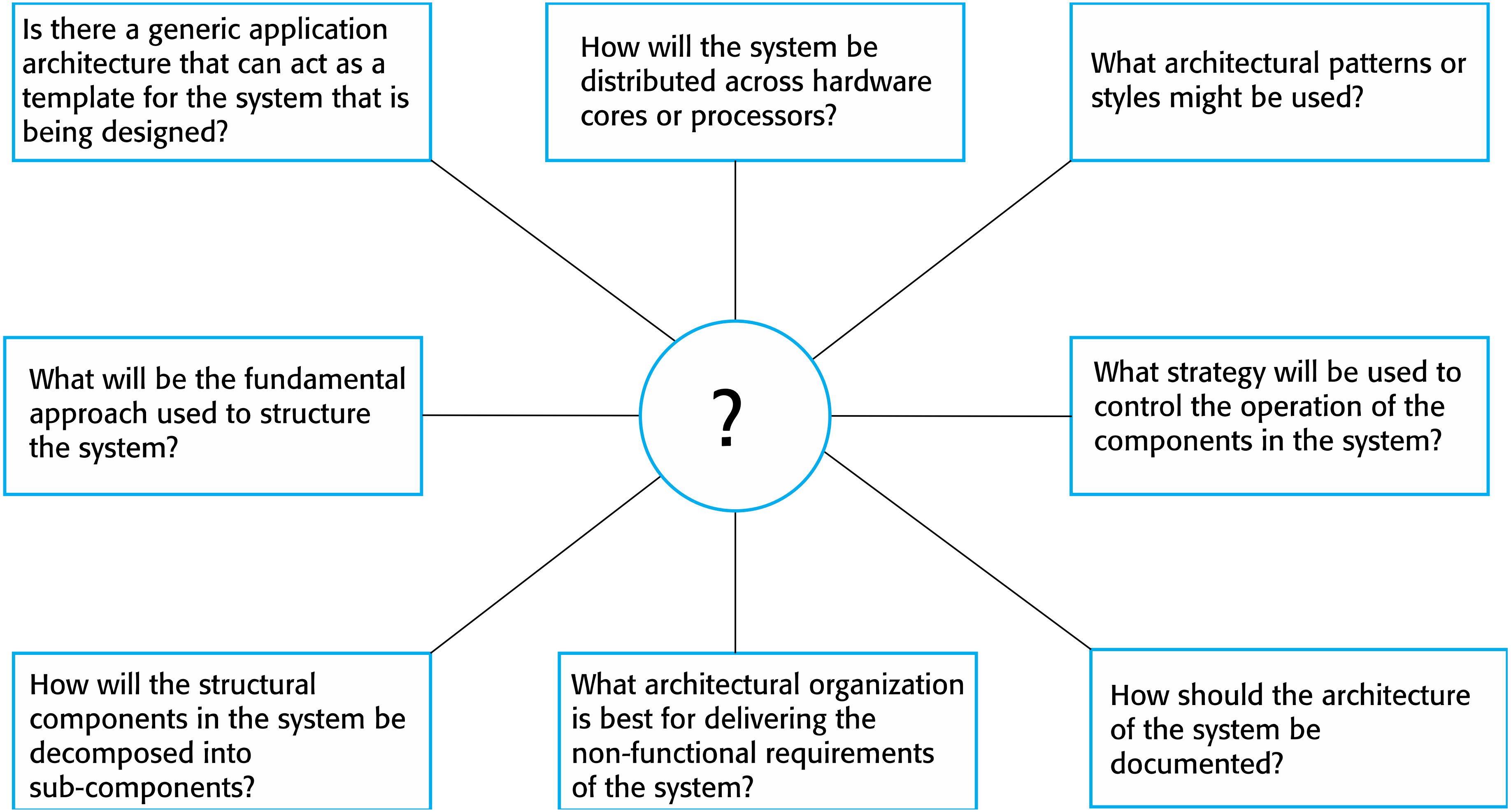
- As a way of **facilitating discussion** about the system design
  - A **high-level architectural view** of a system is useful for communication with **system stakeholders** and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an **abstract view** 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



Is there a generic application architecture that can act as a template for the system that is being designed?

How will the system be distributed across hardware cores or processors?

What architectural patterns or styles might be used?

What will be the fundamental approach used to structure the system?

?

What strategy will be used to control the operation of the components in the system?

How will the structural components in the system be decomposed into sub-components?

What architectural organization is best for delivering the non-functional requirements of the system?

How should the architecture of the system be documented?

Architecture and system characteristics

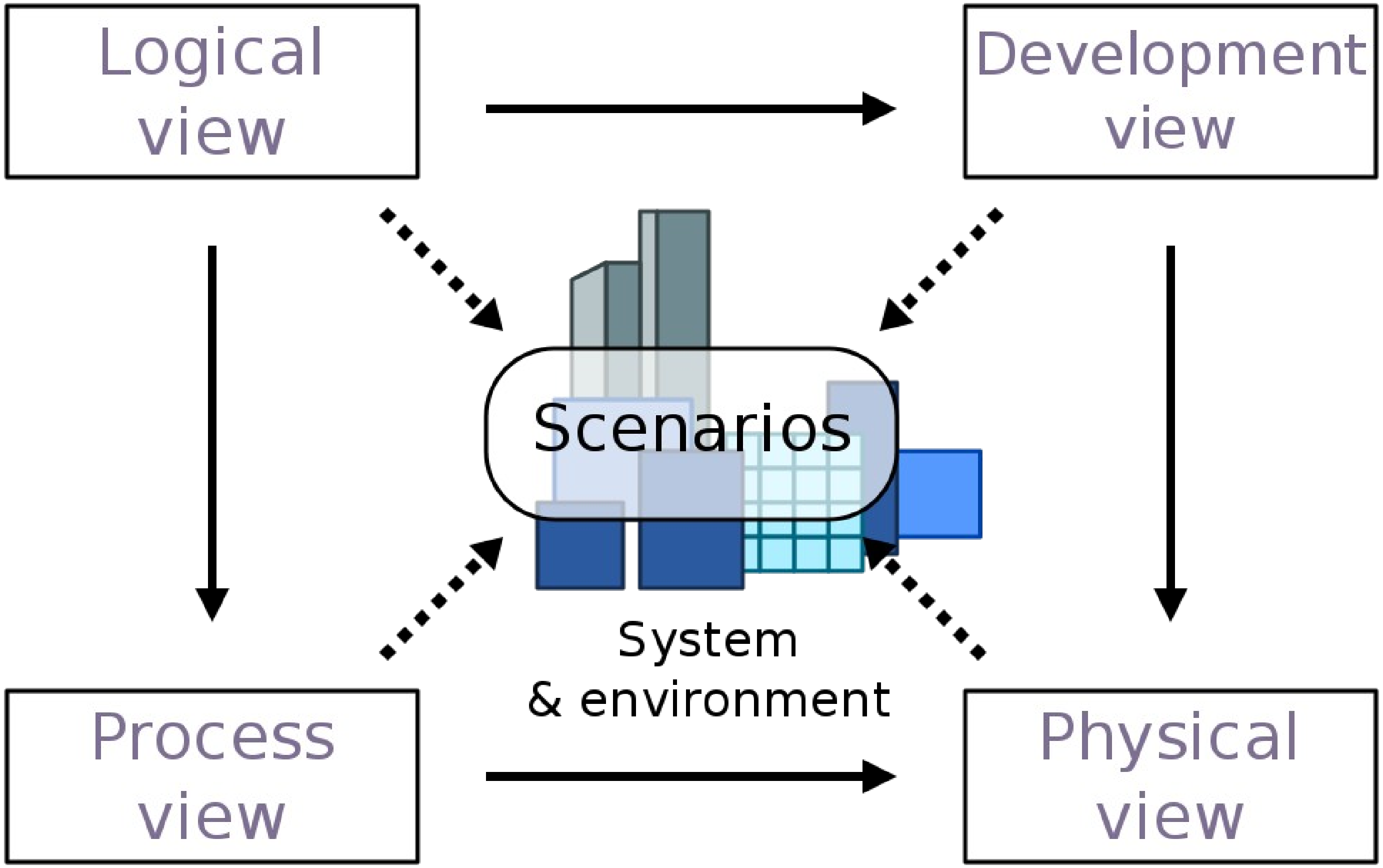
- **Performance**
  - **Localise** critical operations and **minimise** communications. Use **large** rather than **fine-grained components**.
- **Security**
  - Use a **layered architecture** with critical assets in the **inner layers**.
- **Safety**
  - **Localise** safety-critical features in a small number of **sub-systems**.
- **Availability**
  - Include **redundant components** and mechanisms for **fault tolerance**.
- **Maintainability**
  - Use **fine-grained**, **replaceable** components.

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.

Architectural views

Architectural views



https://commons.wikimedia.org/wiki/File:4%2B1_Architectural_View_Model.svg

Architectural views

- What **views** or **perspectives** are useful when designing and documenting a system’s architecture?
- **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

- Different **stakeholders** are interested in **different aspects** of the software.
- The **multiple views** help in looking at the architecture from **different perspectives**.
- End users, developers, system administrators, ...

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 **use cases** or scenarios (+1)

4 + 1 view model of software architecture

Logical view

- Stakeholders: End user
- Aspects addressed: Functional requirements, services, components, relationships, interactions, …
- UML diagrams: Class, state, object, sequence

Process view

- Stakeholders: System integrators
- Aspects addressed: Run-time behavior, dynamic aspects, concurrency, synchronization, …
- UML diagrams: Activity

4 + 1 view model of software architecture

Development view

- Stakeholders: Programmers, developers
- Aspects addressed: Implementation, tools, libraries, execution environment, …
- UML diagrams: Component, package

Physical view

- Stakeholders: System engineers
- Aspects addressed: Hardware, devices, network, …
- UML diagrams: Deployment

4 + 1 view model of software architecture

Use case / scenarios view

- Stakeholders: All
- Aspects addressed: High level requirements, …
- UML diagrams: Use case

Representing architectural views

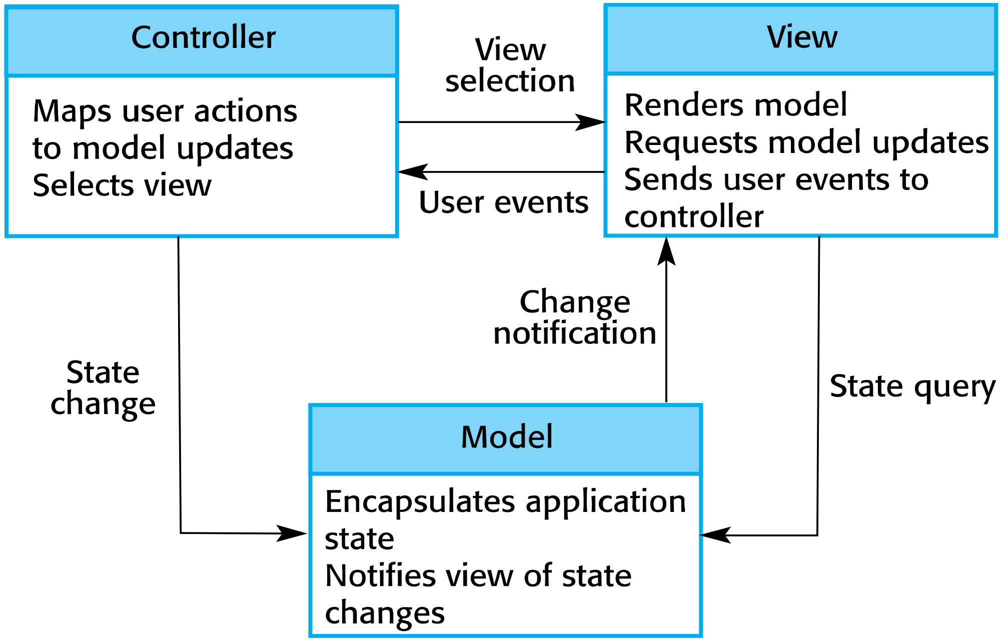
- Unified Modeling Language (**UML**) is an appropriate notation for describing and documenting system architectures (except for high-level system description).
- **UML diagrams**:
  - **Logical view**: Class, state, object, sequence.
  - **Process view**: Activity.
  - **Development view**: Component, package.
  - **Physical view**: Deployment.
  - **Use cases / scenarios**: Use case.

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**.

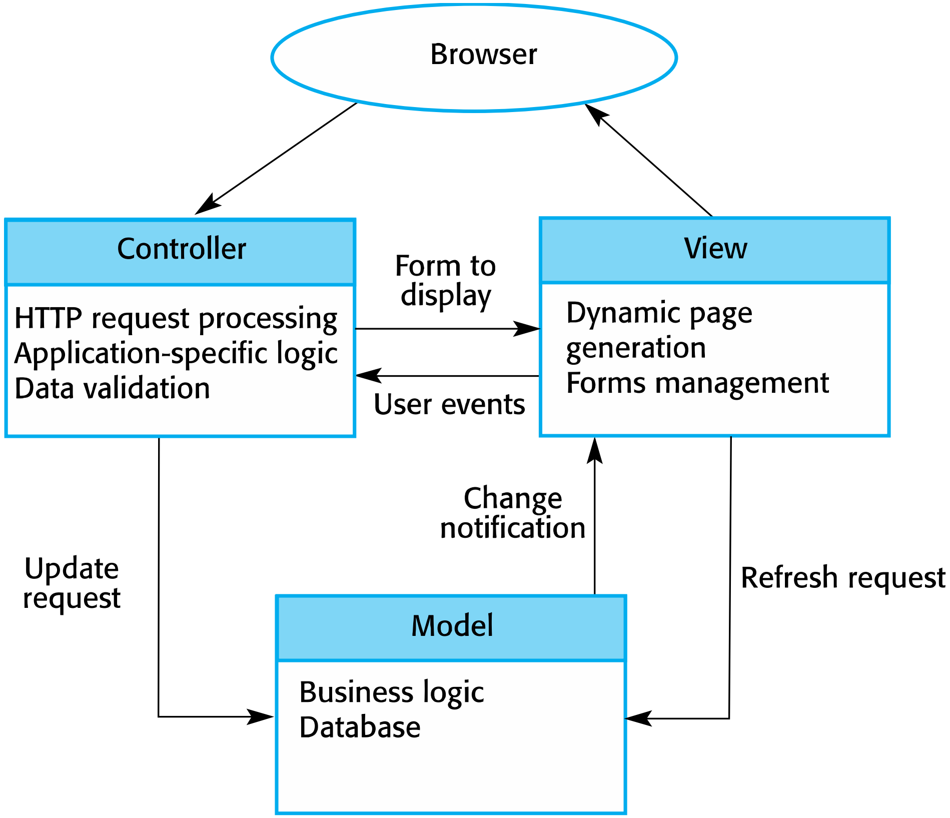
The organization of the Model-View-Controller



The Model-View-Controller (MVC) pattern

Name	MVC (Model-View-Controller)
Description	• 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**.
When used	• 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.
Advantages	• 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.
Disadvantages	• Can involve additional code and code complexity when the data model and interactions are simple.

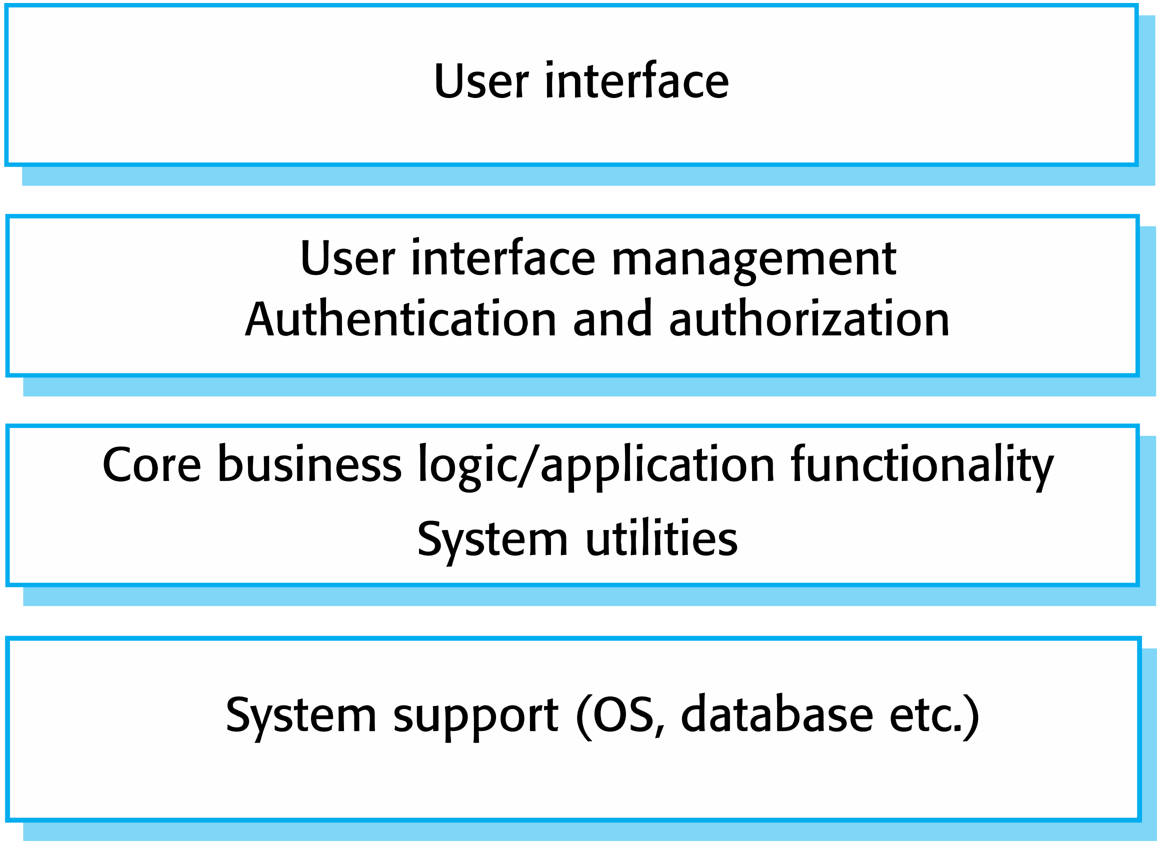
Web application architecture using the MVC pattern



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**.

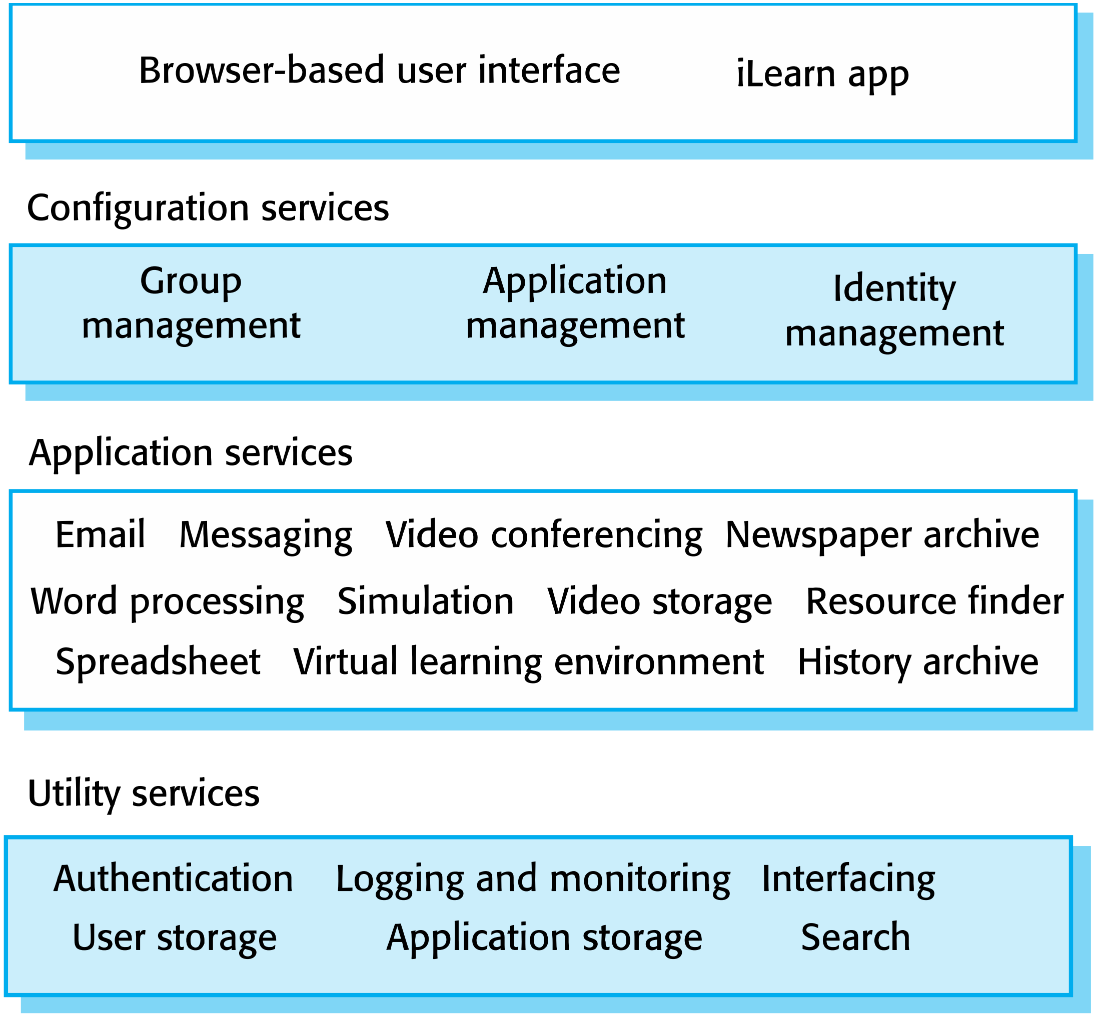
A generic layered architecture


User interface
User interface management
Authentication and authorization
Core business logic/application functionality
System utilities
System support (OS, database etc.)

The 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.
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.

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

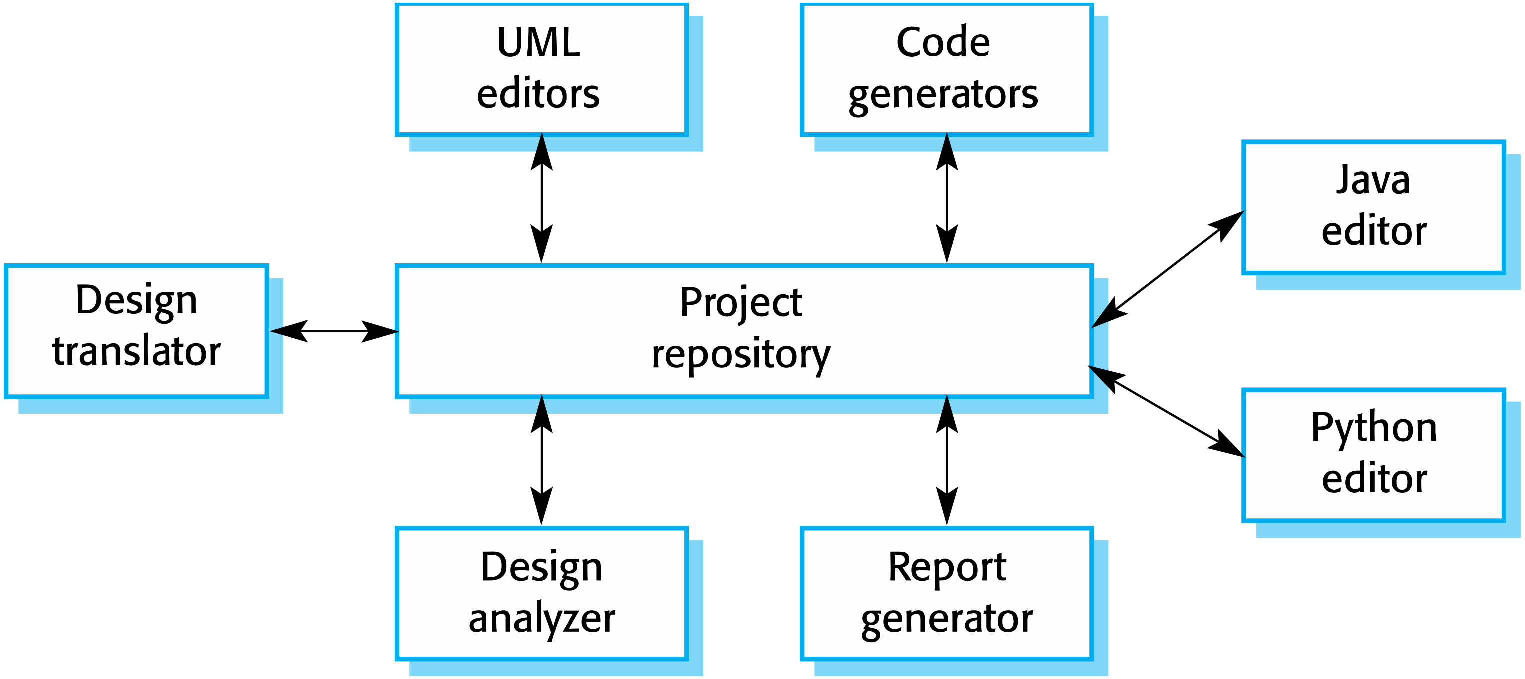
Application storage

Search

Repository architecture

- S**ub-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 a this is an **efficient** data sharing mechanism.

A repository architecture for an IDE



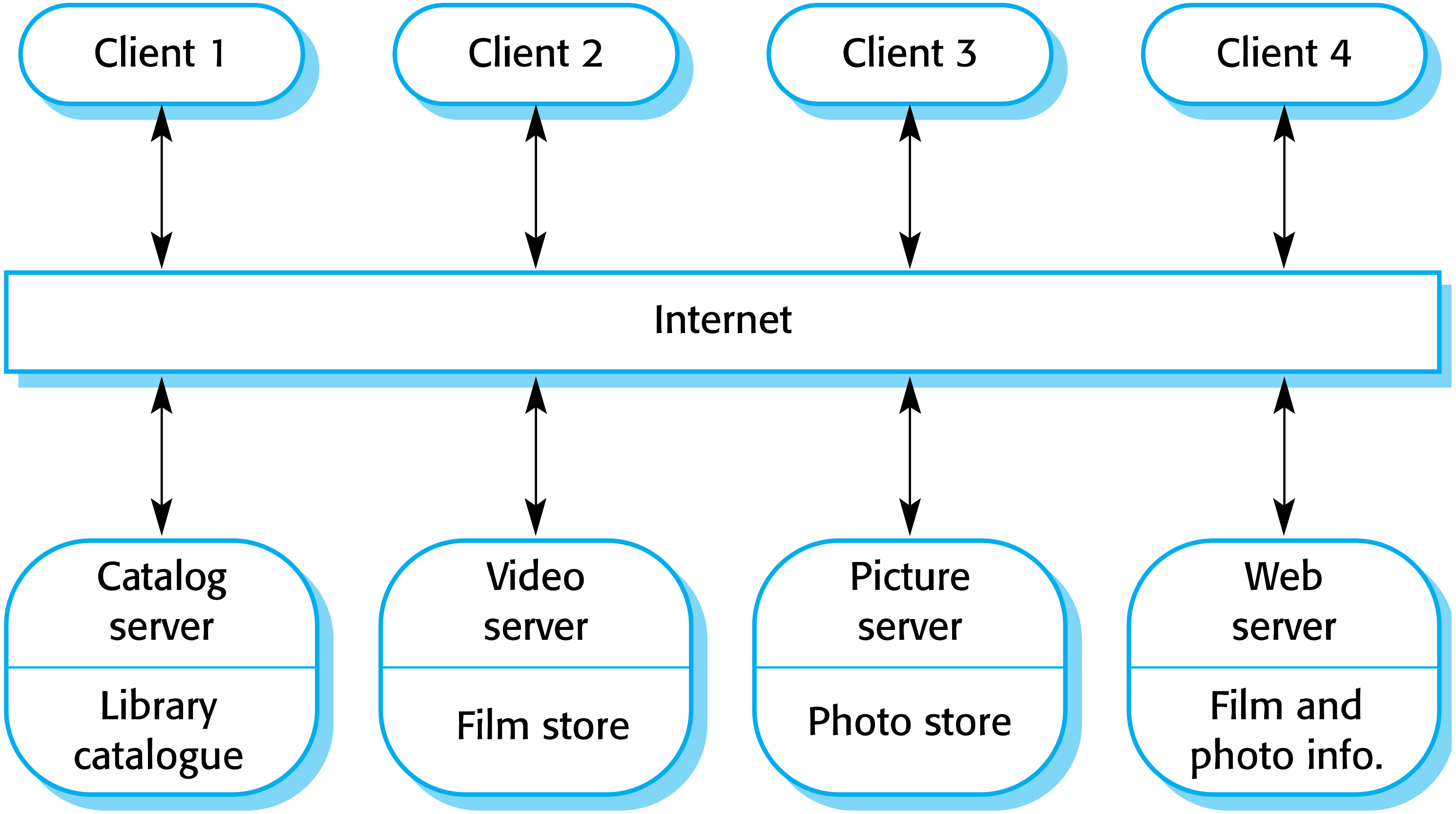
The Repository pattern

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	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.

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.

A client–server architecture for a film library



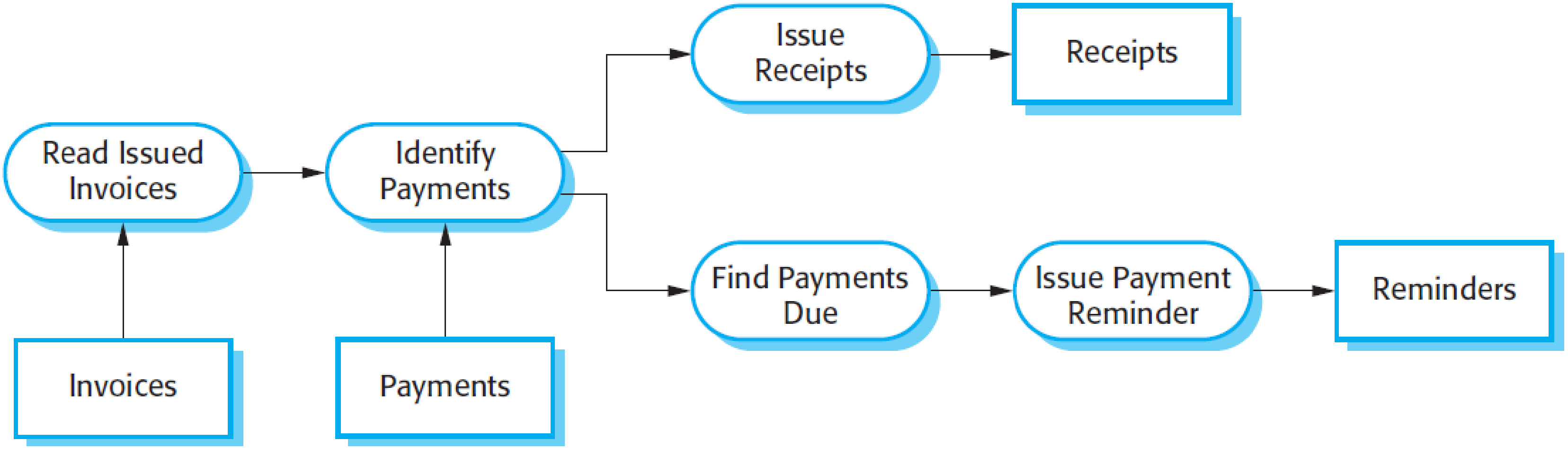
The Client–server pattern

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	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**.

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**.

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



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	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**.