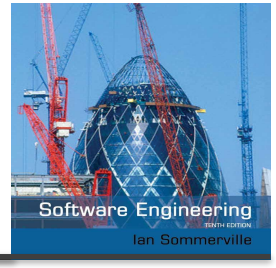
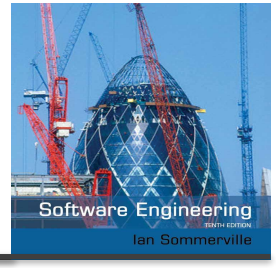


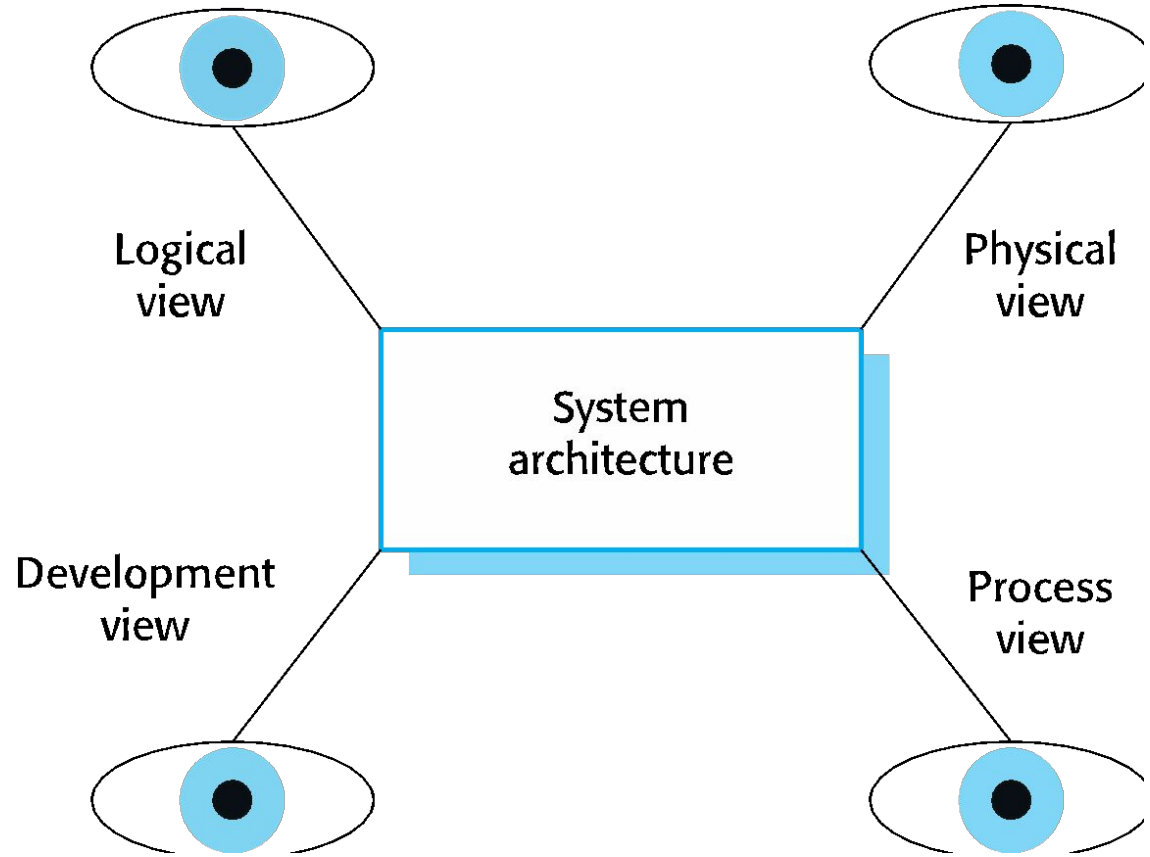
Architectural views

Architectural views



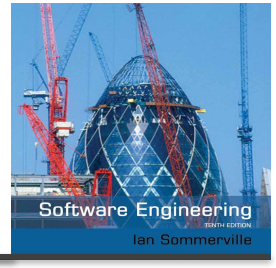
- **What views or perspectives are useful when designing and documenting a system's architecture**?
- **What notations should be used for describing architectural models**?
- **Impossible to show all perspectives from one diagram.**
- So **presenting multiple views to show different perspectives** like:
  - how system is decomposed,
  - how runtime processes interact,
  - how components are distributed across network etc.

Architectural views



4 + 1 view model of software architecture

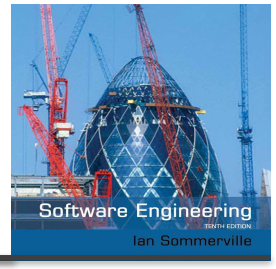
(Krutchen, 1995)



- Proposed 4 basic views linked via different use cases:
- A logical view:
  - **shows the key abstractions in the system as objects or object classes.**
  - Relate **system requirements to entities**
- A process view:
  - shows **how system interacts with interacting processes at run-time**.
  - Useful for **making judgments related to NFRs like availability/performance.**

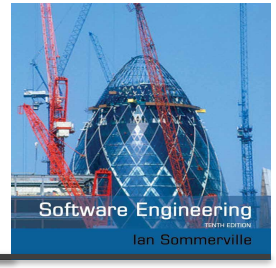
4 + 1 view model of software architecture

(Krutchen, 1995)

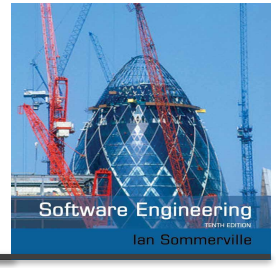


- A development view:
  - **shows breakdown of software assigned to a development team/developer.**
  - Useful for **project managers & developers.**
- A physical view:
  - Shows **how the system hardware and software components are distributed across the processors in the system.**
  - Useful for **system engineers**.
- (Hofmeister et al.,2000) just added conceptual view to this 4 + 1 model.
- Conceptual view: (almost always developed in design process)
  - **Abstract view of high level req. to identify product line arch for future reuse**.
  - Decide which components could be reused for any other application

Representing architectural views

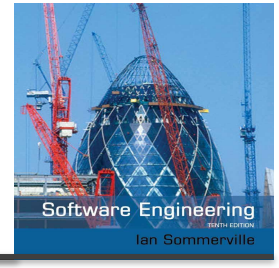


- Use UML for architecture representation?
  - Some people argue that the Unified Modeling Language (UML) is an appropriate notation for describing and documenting system architectures
  - But **UML is designed for object oriented systems**, that will make it more **implementation oriented** rather than being abstract. Moreover, **it'll increase time.**
  - But **UML is of more value when you describe system in detail**.
- Architectural description languages (**ADLs**) have been developed but are not widely used because they are **too domain specific**. **They enforce rules & guidelines for different architectures.**
- Better to go with Informal box /UMLs diagrams for architecture design .



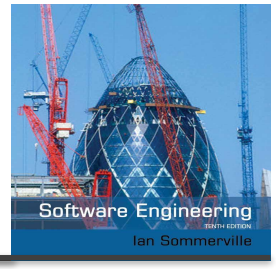
Architectural patterns

Architectural patterns



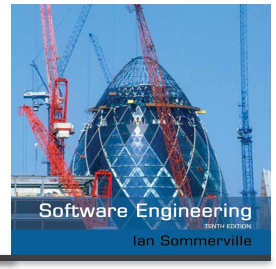
- **Patterns:**
  - **a way of representing, sharing and reusing knowledge about software systems**.
  - described using narrative descriptions or diagrams.
  - Must include the **pattern's strength and weaknesses.**
  - **Pattern description** include:
    - **Pattern name**
    - **Brief description**
    - **Graphical model**
    - **Example of a relevant system** where that pattern have already been used.
    - When to use that pattern , **pros & cons of the pattern**.
- The fundamental idea behind arch. is separation and independence. Because it is beneficial for incorporating the localized changes only.
- We will discuss some widely used architectural patterns.

MVC



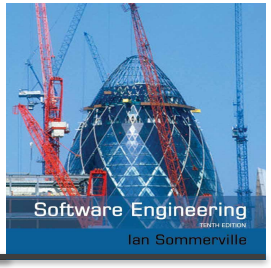
- **Split application into multiple sections** all have own purpose
- User sent request to controller
- Controller being the middle man between model and view, process the client request and sent to Model
- Model handles data logic, access database and do authentication/update/delete or whatever is needed to perform on data.
- Controller pass the raw information received from model to view
- View presents the data
- Return to controller and controller send response to client
- This is a way to segregate data presentation from data logic.

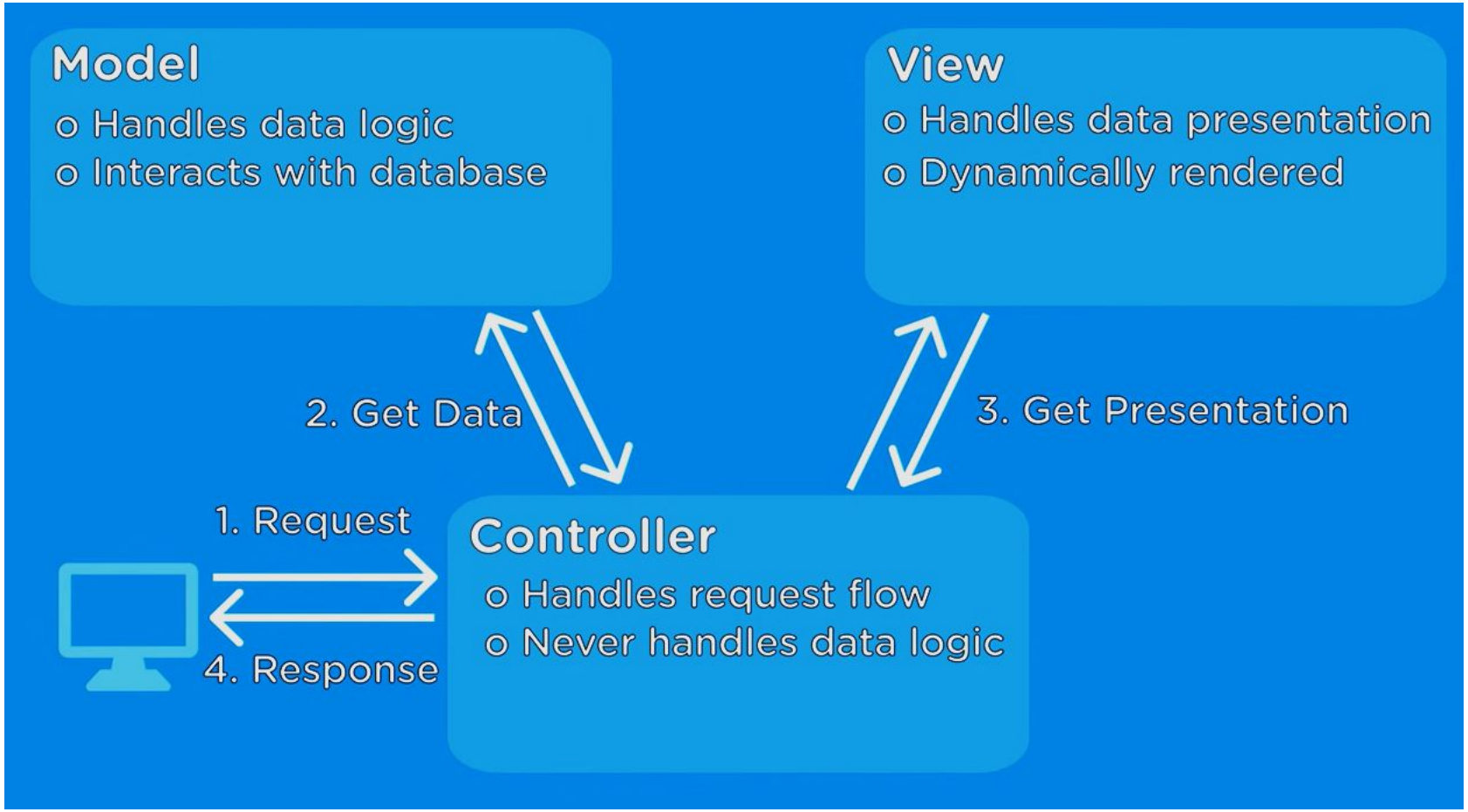
The Model-View-Controller (MVC) pattern



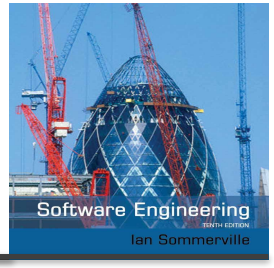
Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. MVC divides application into 3 parts. (Separation of concerns) • The Model component manages the system data and associated operations on that data. • The View component defines and manages how the 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.
Example	A figure in successive slides shows the architecture of a **web-based application system** organized using the MVC pattern.
When used	Used **when:** • **there are multiple ways to view and interact with data.** • **the future requirements for interaction and presentation of data are unknown.**
Advantages	• Allows the data to change independently of its representation and vice versa. (**adding a new view or changing an existing one might not affect the underlying data in model**) • Supports multiple presentation of the same data (changes made in one representation reflected in all of them.)
Disadvantages	**Can involve additional code and increase code complexity.**

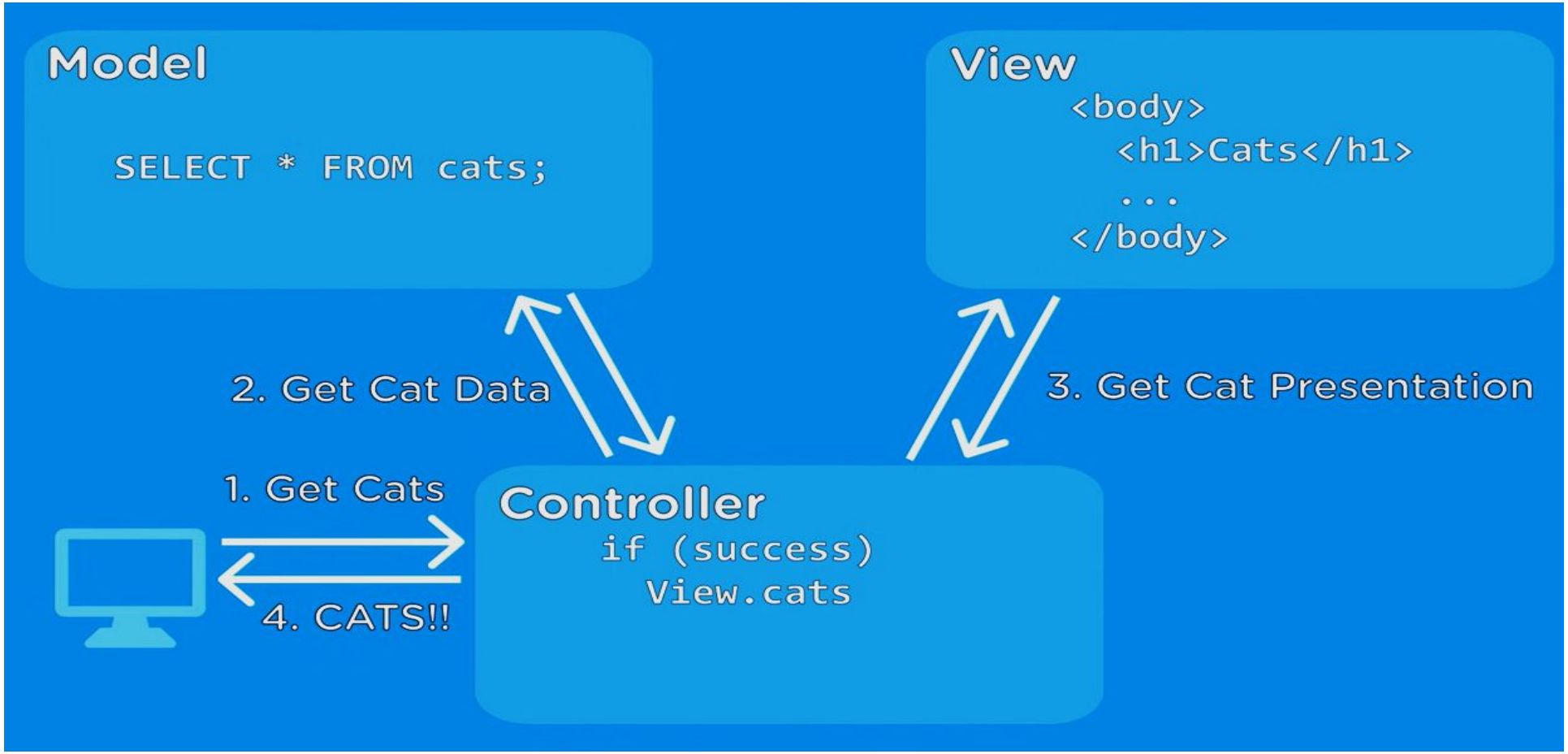
The organization of the Model-View-Controller



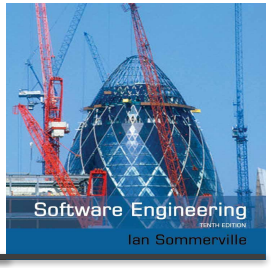


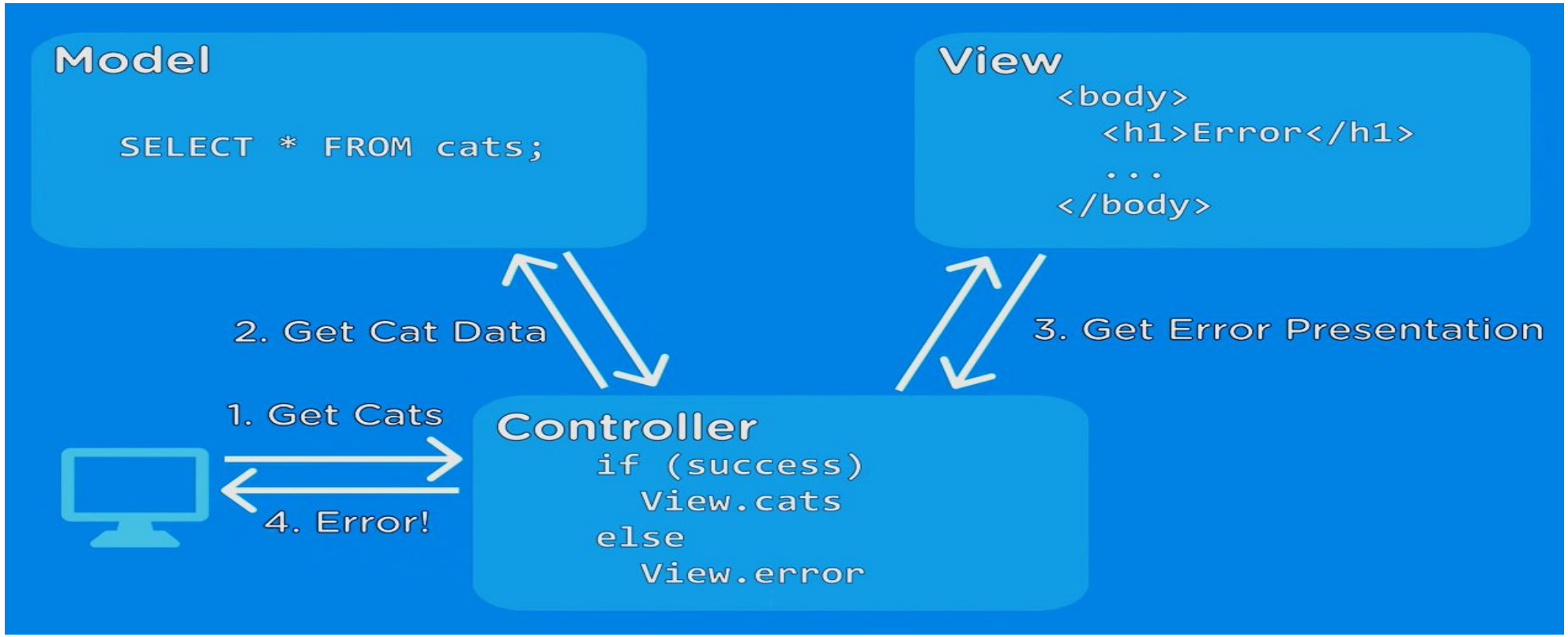
Example of MVC based web application



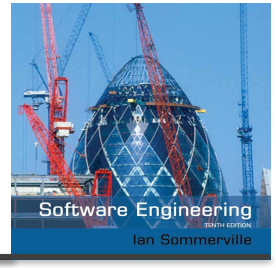


Example of MVC based web application



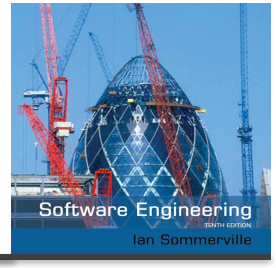


Layered architecture



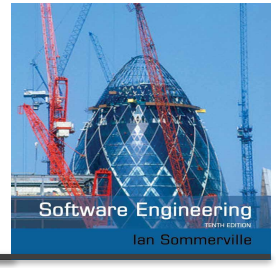
- **Organises the system functionality into a set of layers each of which provide a set of services.**
- Typically **layers are UI, Auth & Authorization, Business logic, OS / DB**
- **Every layer relies on** facilities and services offered **by the layer immediately beneath it.**
- Upper layer talk to layer below that, **no jumping between layers is allowed**
- **Lower layers never called to upper layer** they just replied to the inputs from upper layer
- Requesting something from a layer means **iterating down to multiple layers**
- **Structuring tasks into teams is easy** as multiple teams may work for different layers. (**supports incremental development**)

Layered architecture



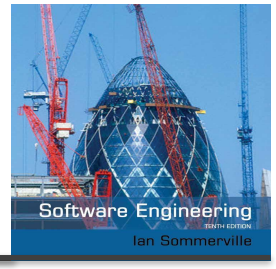
- Advantages:
  - Easy to embed NFRs like security by inserting a layer without disrupting the whole system i.e., **build on top approach**
  - When a **layer is updated**, then only the **adjacent layer is affected.**
  - **Easy to support multi platform applications as only the machine dependent layers need some changes to support multi OS or databases.**
  - Module replacement or upgradation is easy
- Disadvantages:
  - **performance is an issue as a request has to go over multiple layers for a reply**
  - Its really **difficult to code, like avoid going to specific layers (creating this level of independence between layers)**.

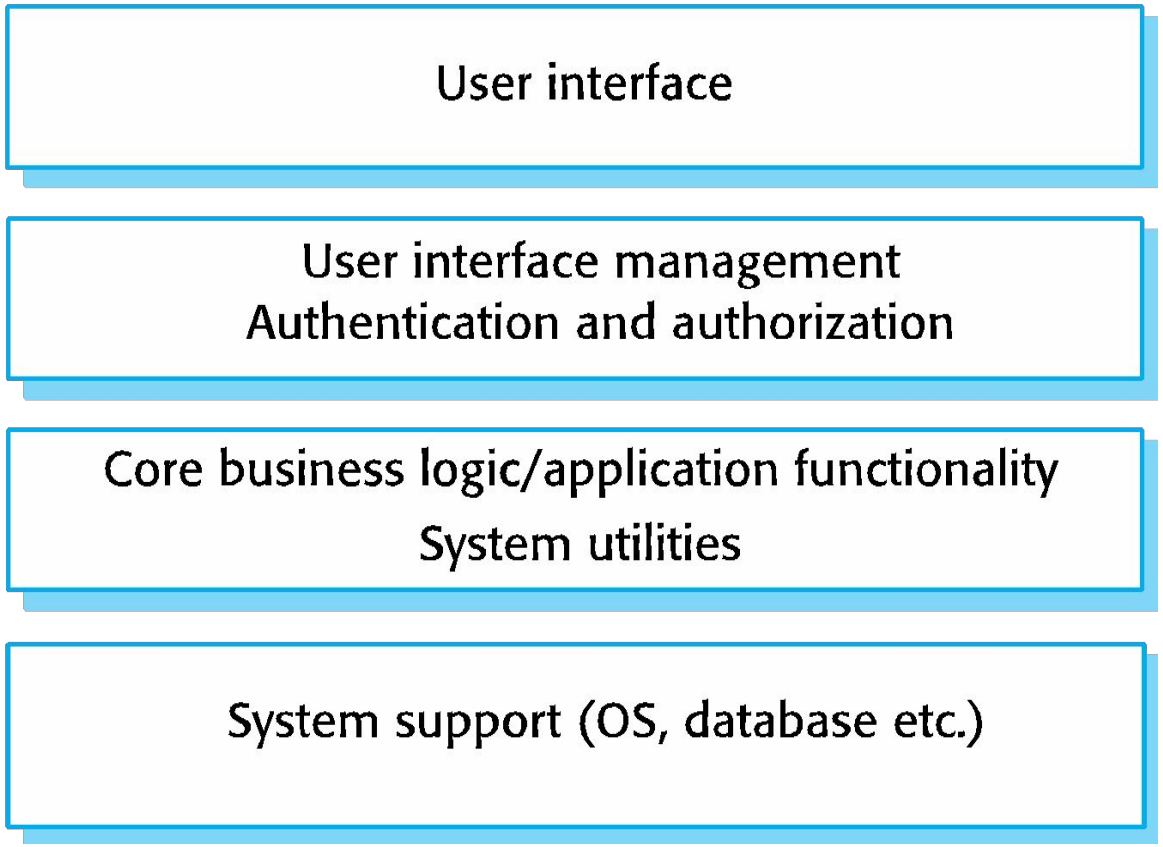
The Layered architecture pattern



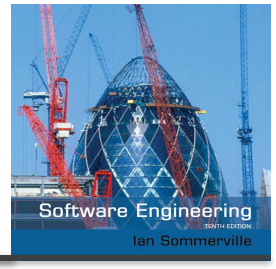
Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.
Example	A layered model of a system for sharing copyright documents held in different libraries.
When used	Used when: • **building new facilities on top of existing systems**; • **development is spread across several teams** with each team responsibility for a layer of functionality; • 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.

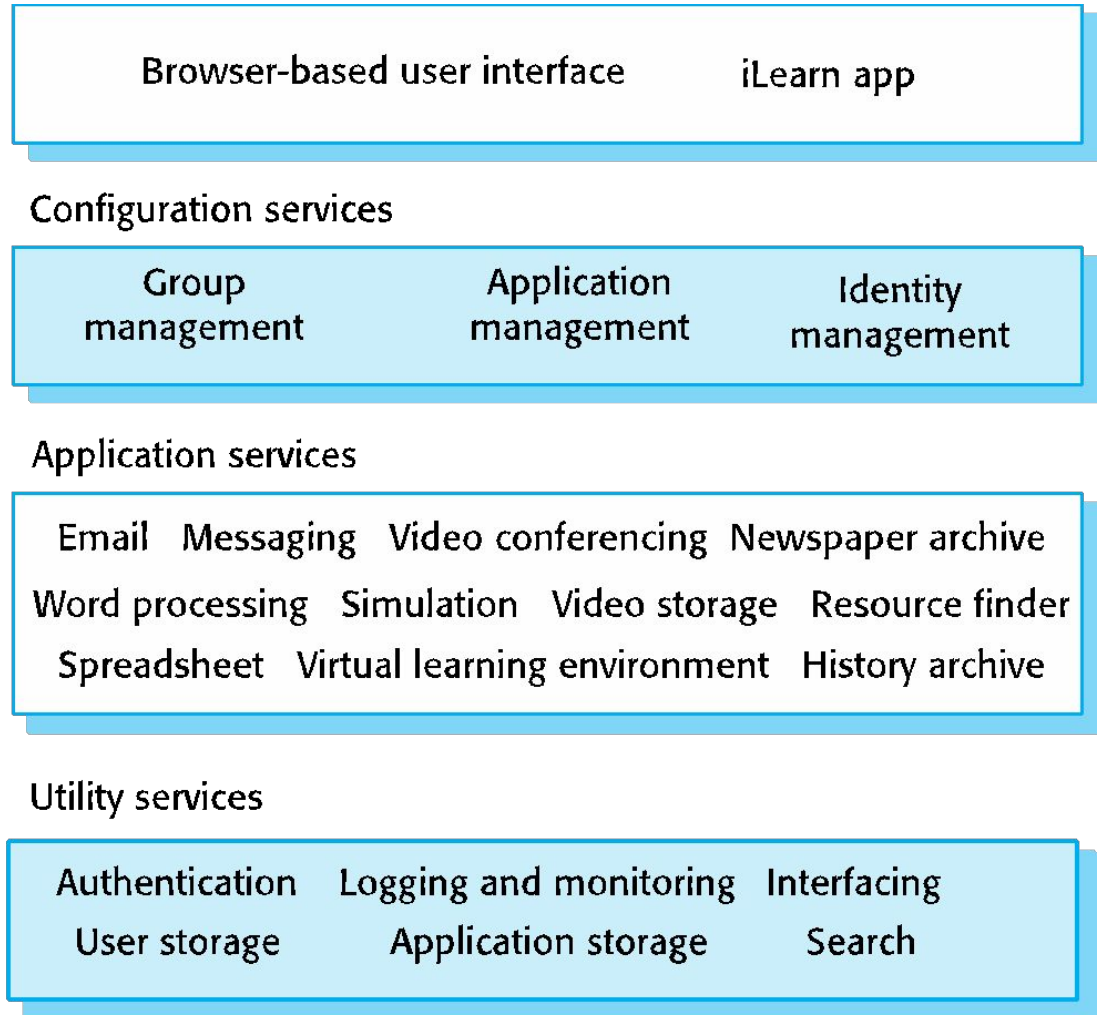
A generic layered architecture



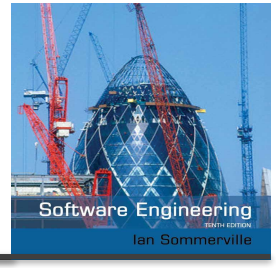


The architecture of the iLearn system case study (covered in lecture 2)



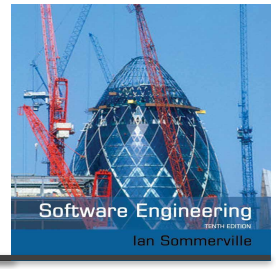


Repository architecture



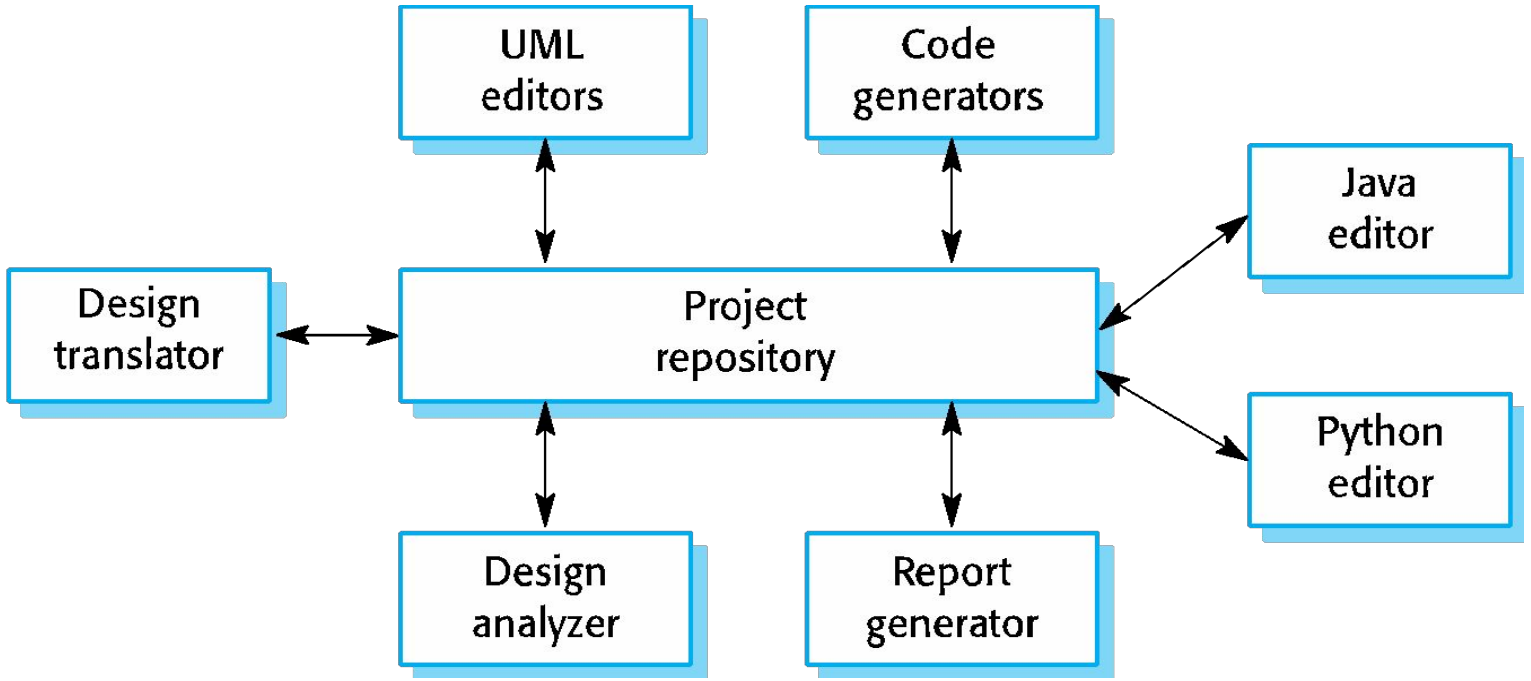
- **Sub-systems must exchange data**.
- This may be done in two ways:
  - **Shared data is held in a central database or repository** and may be accessed by all sub-systems;
  - **Each sub-system maintains its own database and passes data explicitly** to other sub-systems.
- When **large amounts of data** are to be shared, the **repository model** of sharing is most commonly used a this is an efficient data sharing mechanism.
- It's a static structure, doesn't change at runtime. (difference from client-server architecture).

The Repository pattern

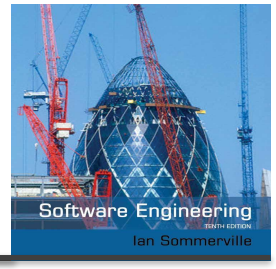


Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Examples are: Interactive Development Environment for softwares, MIS etc.
When used	When: • large volumes of information are generated that has to be stored for a long time. • 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) ~ central repository
Disadvantages	• The **repository is a single point of failure** so problems in the repository affect the whole system. • **Distributing the repository across several computers may be difficult**.

A repository architecture for an IDE

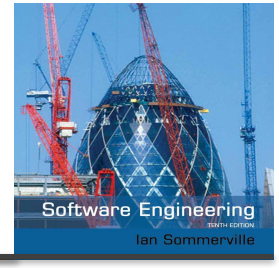


Client-server architecture



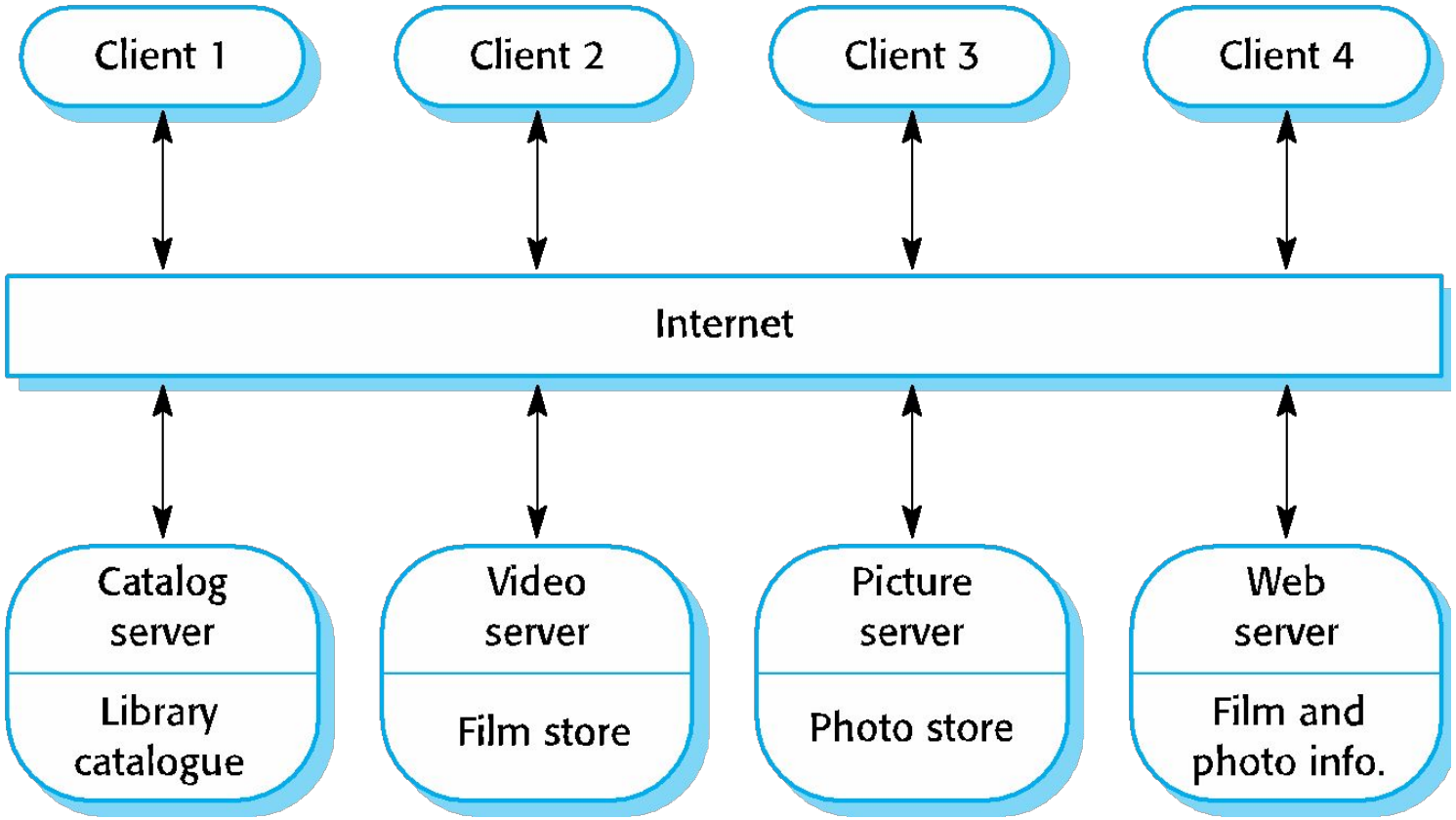
- Runtime organization of distributed systems
- A set of services mounted over associated servers and client access the services via servers.
- Components:
  - **A set of Servers**: print server, file server, management server etc.
  - **A set of clients**: different instances of a client program running concurrently on multiple machines to access service offered by server.
  - **Network**: distributed system architecture using internet protocols
- Client access services offered by servers via remote procedure invocation using request-reply protocols.

The Client–server pattern

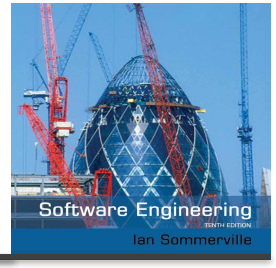


Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Next figure shows example of a film and video/DVD 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. • May also be used when the load on a system is variable.
Advantages	• Resource sharing • Independent servers • No single point failure • transparent system upgradation
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

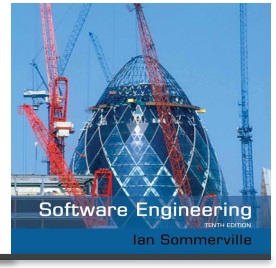


Pipe and filter architecture



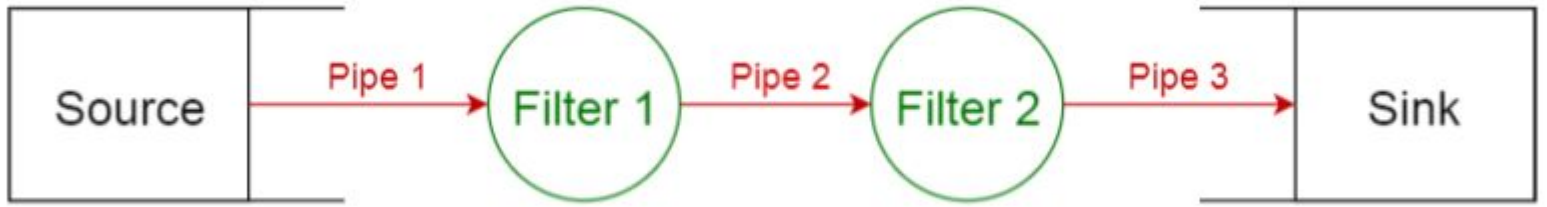
- Functional transformations process their inputs to produce outputs (that's why its called as filter).
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- **Best for batch processing, embedded systems or systems with limited user interactions.**
- **Not really suitable for interactive systems** because the input data stream needs to be pre-processed. GUI based I/O systems have complex input streams like mouse click events, for which converting into sequential inout stream is challenging.

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	Coming figure 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. • Compilers. The consecutive filters perform lexical analysis, parsing, semantic analysis, and code generation
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 unparsed 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.

pipe and filter architecture



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