# Architectural design

Lecture 21

# Agenda

- understand why the architectural design of software is important;
- understand the decisions that have to be made about the system architecture during the architectural design process;
- some architectural styles covering the overall system organisation, modular decomposition and control;
- understand how reference architectures are used to communicate architectural concepts and to assess system architectures.

# Architectural Design

- Large systems are always decomposed into sub-systems that provide some related set of services.
- Design process of identifying these sub-systems and establishing a framework for sub-system control and communication is called architectural design.
- The output of this design process is a description of the software architecture.

# Why explicitly designing a software architecture ?

- *Stakeholder communication*
- *System analysis*
- *Large-scale reuse*

# *Stakeholder communication*

- The architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.

# *System analysis*

- Making the system architecture explicit at an early stage in the system development requires some analysis.
- Architectural design decisions have a profound effect on whether the system can meet critical requirements such as performance, reliability and maintainability.

# *Large-scale reuse*

- A system architecture model is a compact, manageable description of how a system is organised and how the components interoperate.
- The system architecture is often the same for systems with similar requirements and so can support large-scale software reuse.

# Helps consider key design aspects

- Software architecture can serve as a design plan that is used to negotiate system requirements and as a means of structuring discussions with clients, developers and managers.
- essential tool for complexity management. It hides details and allows the designers to focus on the key system abstractions.

# Architectural Decisions

- The particular style and structure chosen for an application may therefore depend on the non-functional system requirements:
    1. Performance
    2. Security
    3. Safety
    4. Availability
    5. Maintainability

# Performance

- If performance is a critical requirement, the architecture should be designed to localise critical operations within a small number of subsystems, with as little communication as possible between these sub-systems.
- This may mean using relatively large-grain rather than fine-grain components to reduce component communications.

# Security

- If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers and with a high level of security validation applied to these layers.

# Safety

- If safety is a critical requirement, the architecture should be designed so that safety-related operations are all located in either a single sub-system or in a small number of sub-systems.
- This reduces the costs and problems of safety validation and makes it possible to provide related protection systems.

# Availability

- If availability is a critical requirement, the architecture should be designed to include redundant components and so that it is possible to replace and update components without stopping the system.

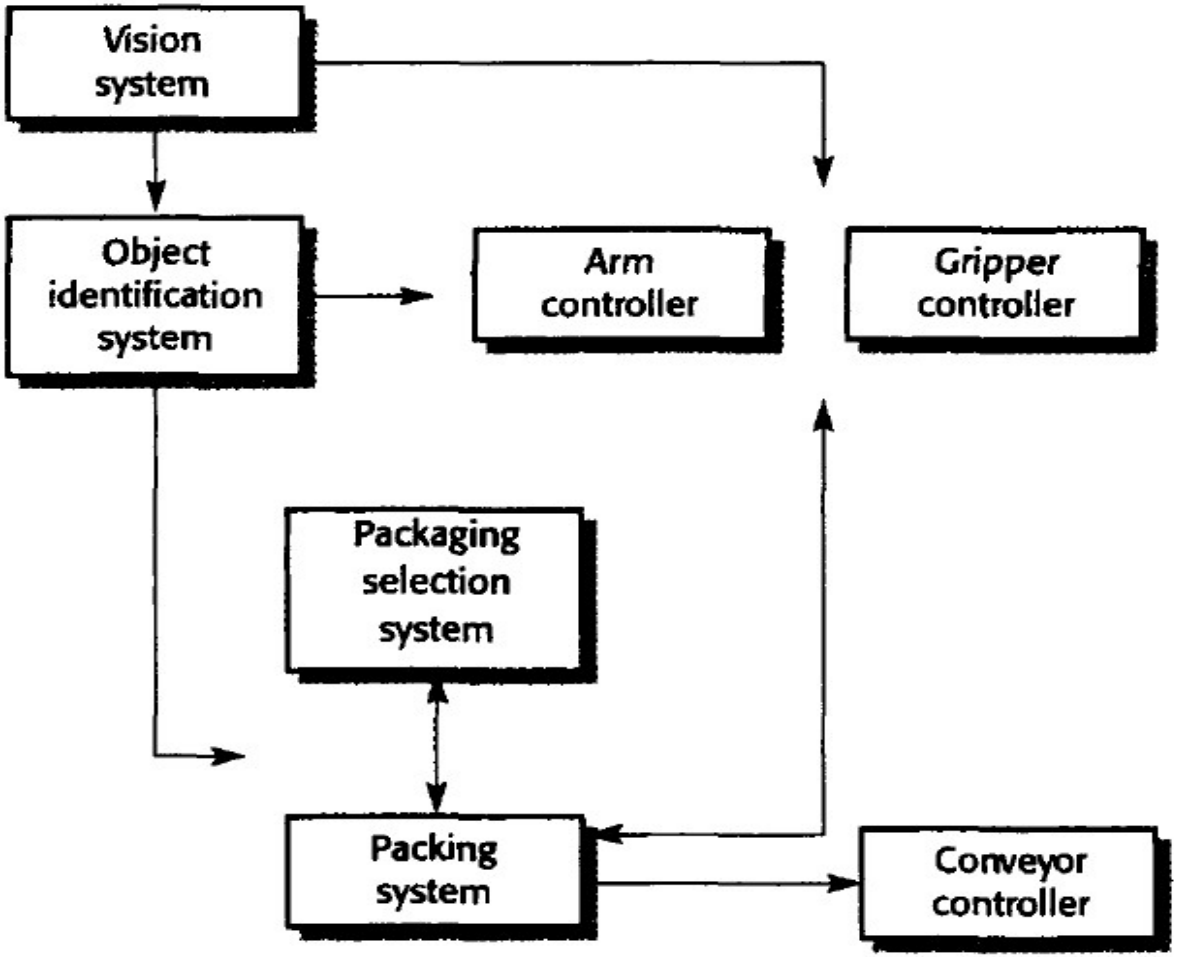
# Maintainability

- If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may readily be changed.
- Producers of data should be separated from consumers and shared data structures should be avoided.

# Conflict of interest

- Potential conflict between some of these architectures.
  - For example, using large-grain components improves performance, and using fine-grain components improves maintainability. If both of these are important system requirements, then some compromise solution must be found

# Architecture for a packing robot 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 a number of processors?
- What architectural style or styles are appropriate for the system?
- What will be the fundamental approach used to structure the system?

# Contd..

- How will the structural units in the system be decomposed into modules?
- What strategy will be used to control the operation of the units in the system?
- How will the architectural design be evaluated?
- How should the architecture of the system be documented?

# Design Document

- The product of the architectural design process is an architectural design document.
  - A *static structural model that shows the sub-systems or components that are* to be developed as separate units.
  - A *dynamic process model that shows how the system is organised into processes* at run-time. This may be different from the static model.

# Design Document

- An *interface model that defines the services offered by each sub-system* through its public interface.
- *Relationship models that shows relationships, such as data flow, between the* sub-systems.
- A *distribution model that shows how sub-systems may be distributed across* computers.

# Architectural Styles

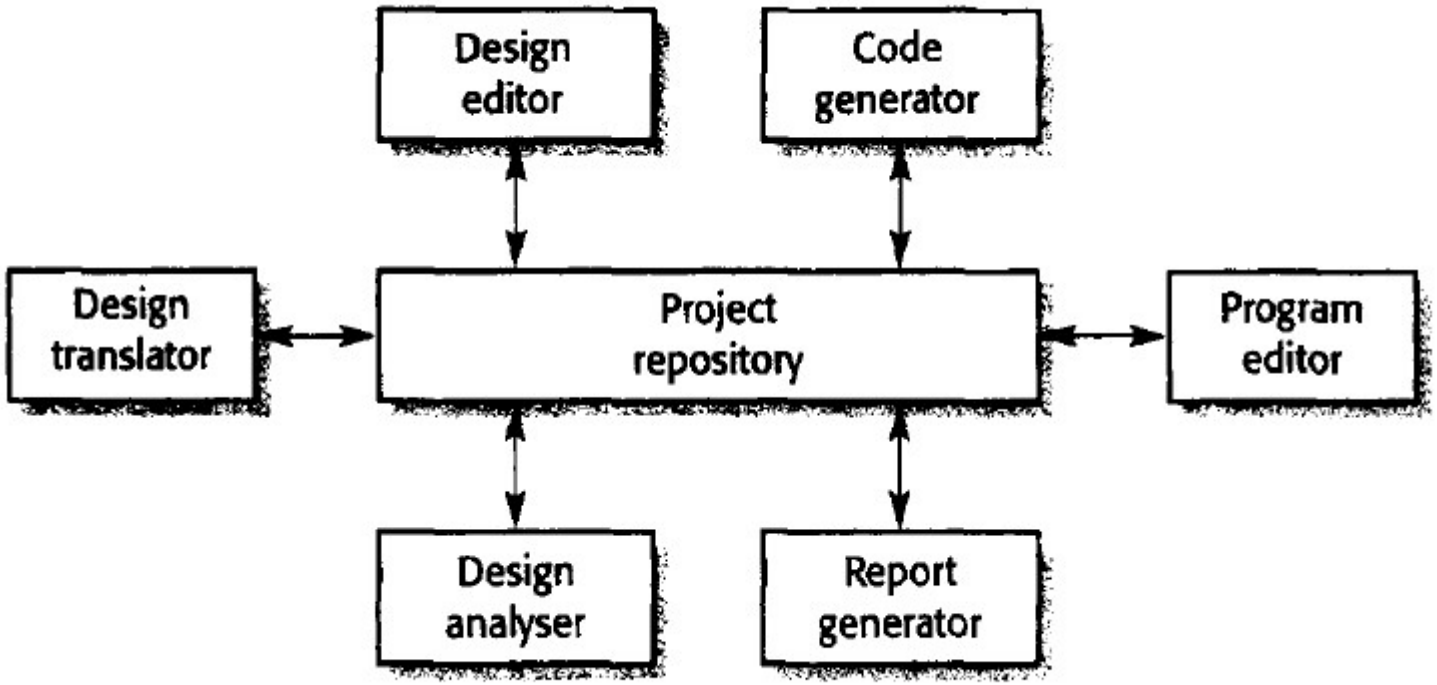
- Repository Model
- Client Server model
- Layered Model

# The repository model

1. All shared data is held in a central database that can be accessed by all sub-systems. A system model based on a shared database is sometimes called a *repository model*.

2. Each sub-system maintains its own database. Data is interchanged with other sub-systems by passing messages to them.

# Architecture CASE Toolset



# Advantages /Disadvantages of a shared repository

- It is an efficient way to share large amounts of data. There is no need to transmit data explicitly from one sub-system to another.
- However, sub-systems must agree on the repository data model. Inevitably, this is a compromise between the specific needs of each tool. Performance may be adversely affected by this compromise. It may be difficult or impossible to integrate new sub-systems if their data models do not fit the agreed schema.

# Advantages /Disadvantages of a shared repository

- Sub-systems that produce data need not be concerned with how that data is used by other sub-systems.
- However, evolution may be difficult as a large volume of information is generated according to an agreed data model. Translating this to a new model will certainly be expensive; it may be difficult or even impossible.

# Advantages /Disadvantages of a shared repository

- Activities such as backup, security, access control and recovery from error are centralized. They are the responsibility of the repository manager. Tools can focus on their principal function rather than be concerned with these issues.
- Different sub-systems may have different requirements for security, recovery and backup policies. The repository model forces the same policy on all sub-systems.

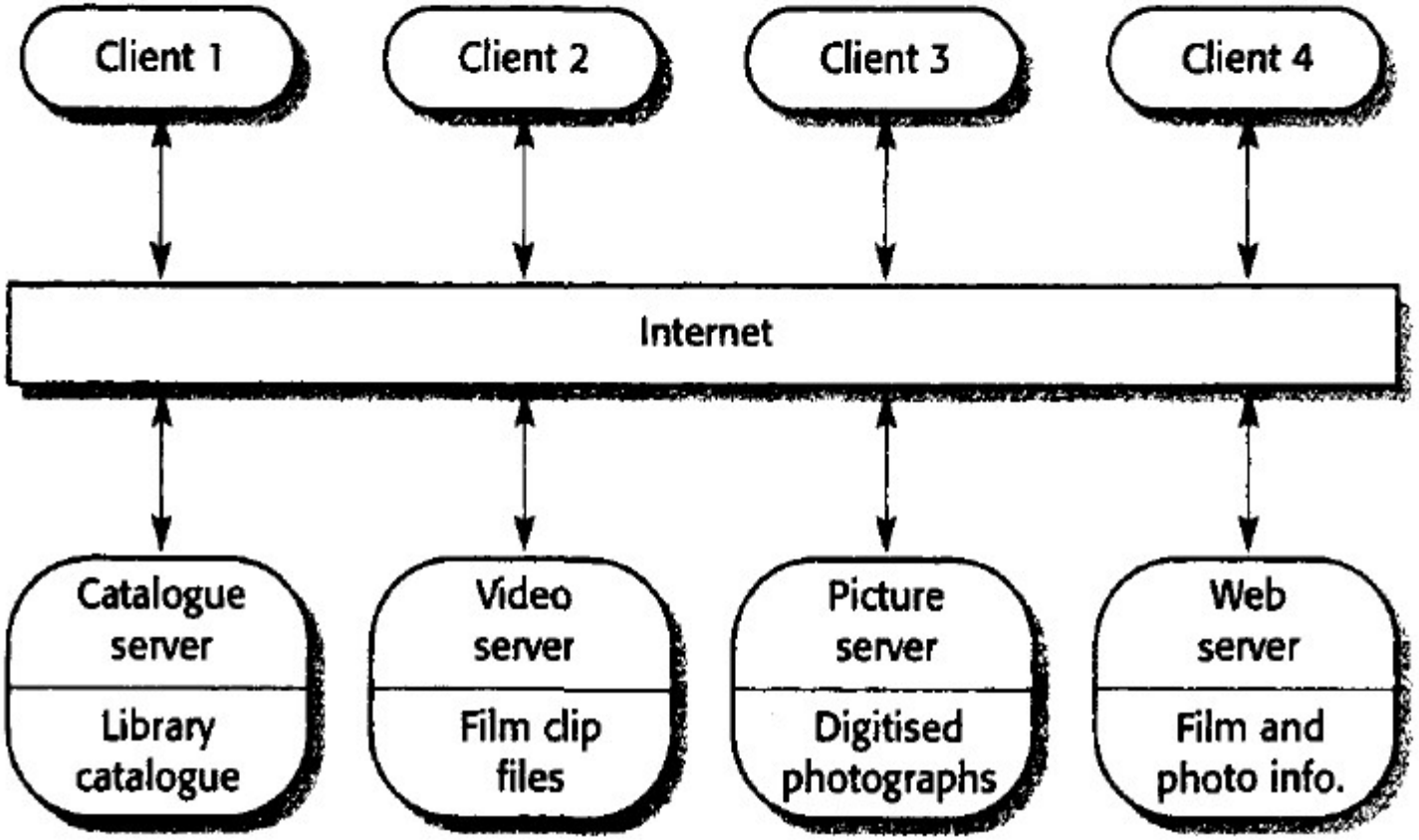
# Advantages /Disadvantages of a shared repository

- The model of sharing is visible through the repository schema. It is straightforward to integrate new tools given that they are compatible with the agreed data model.
- Difficult to distribute the repository over a number of machines. Although it is possible to distribute a logically centralized repository, there may be problems with data redundancy and inconsistency.

# The client-server model

- The client-server architectural model is a system model where the system is organized as 2 set of services and associated servers and clients that access and use the services.

# Film and picture library system



# Advantages

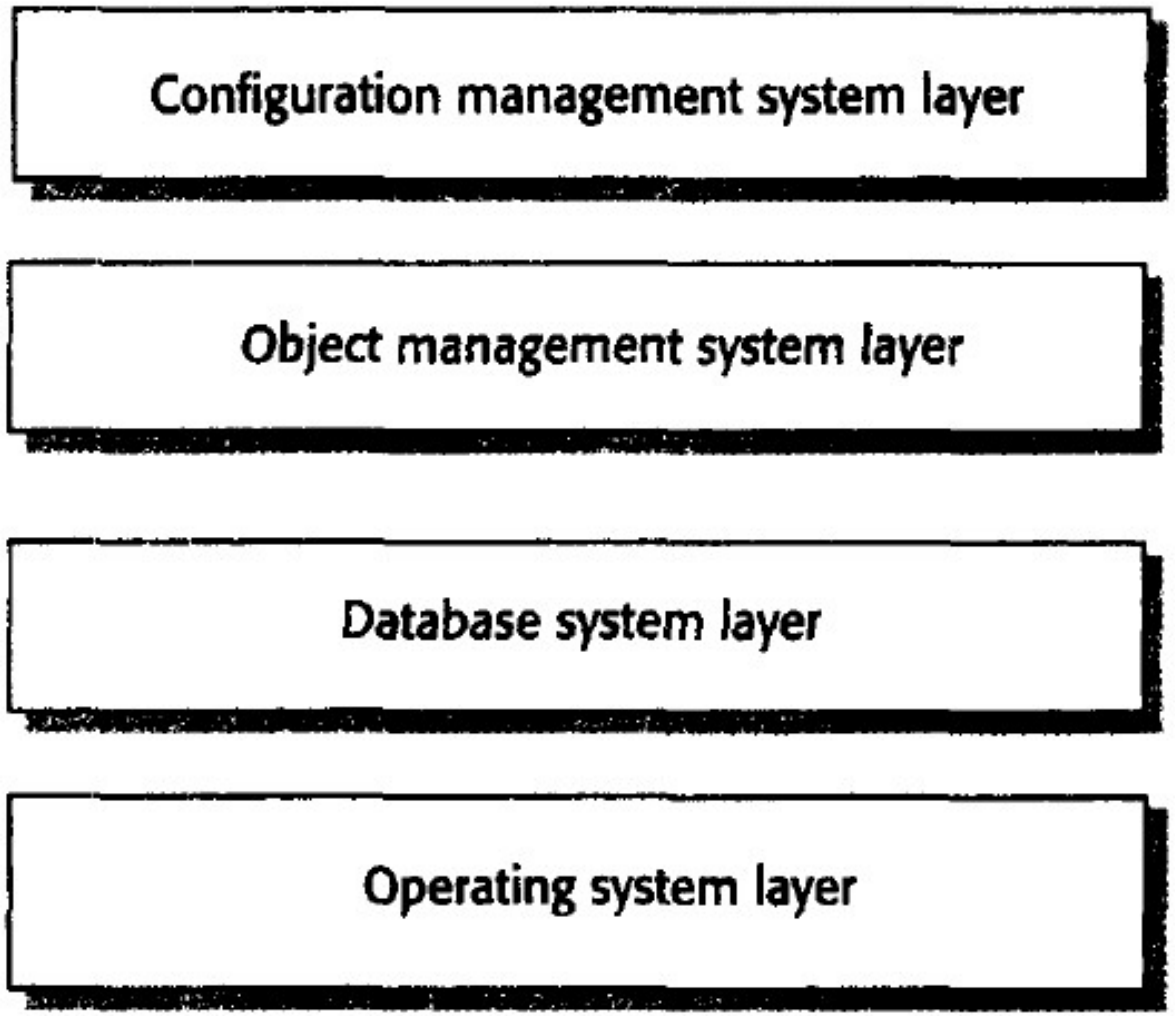
- client-server model is that it is a distributed architecture.
- Effective use can be made of networked systems with many distributed processors.
- It is easy to add a new server and integrate it with the rest of the system or to upgrade servers transparently without affecting other parts of the system.

# The layered model

- The layered model of an architecture (sometimes called an abstract machine model) organizes a system into layers, each of which provide a set of services

- An example of a layered model is the OSI reference model of network protocols

# Layered model of a version management system



# Example

- The configuration management system manages versions of objects and provides general configuration management facilities
- it uses an object management system thaT provides information storage and management services for configuration items or objects.
- This system is built on top of a database system to provide basic data storage and services such as transaction management, rollback and recovery,
- and access control.

# Advantages

- The layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made available to users.
- This architecture is also changeable and portable.
- As layered systems localize machine dependencies in inner layers, this makes it easier to provide multi-platform implementations

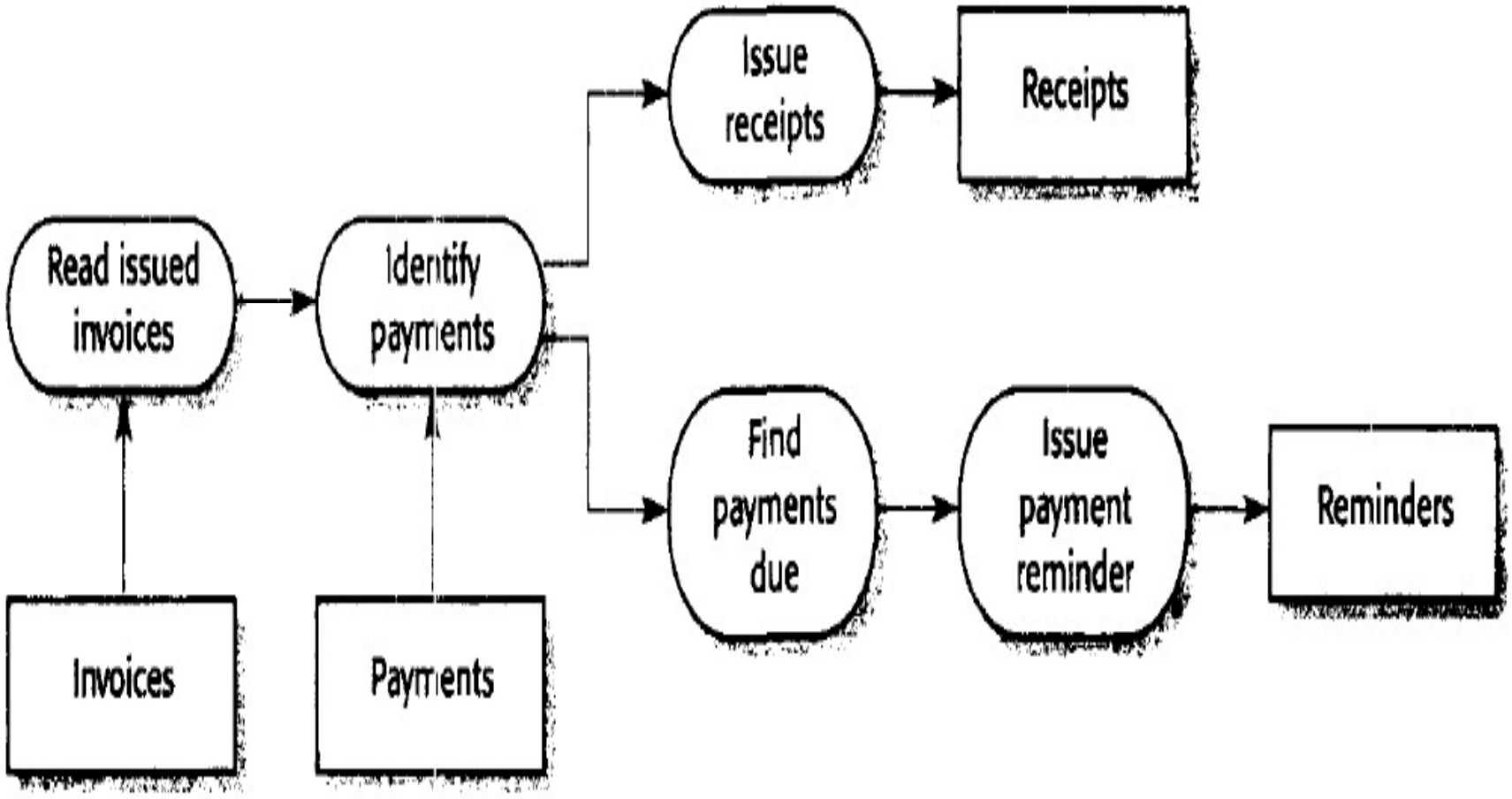
# Disadvantages

- A disadvantage of the layered approach is that structuring systems in this way can be difficult. Inner layers may provide basic facilities, such as file management, that are required at all levels.
- Performance can also be a problem because of the multiple levels of command interpretation that are sometimes required

# Modular decomposition styles

- *Object-oriented decomposition*
- *FUnction-oriented pipelining*

# pipeline model of an invoice processing



# *Advts of Object-oriented decomposition*

- Objects are loosely coupled, the implementation of objects can be modified without affecting other objects.
- Objects are often representations of real-world entities so the structure of the system is readily understandable.
- Object-oriented programming languages have been developed that provide direct implementations of architectural components.

# Advts **Function-oriented pipelining**

- It supports the reuse of transformations.
- It is mtuitive lin that many people think of their work in terms of input and output processing.
- Evolving the system by adding new transformations is usually straightforward.
- It is simple to implement either as a concurrent or a sequential system

# Control styles

- Centralised control One sub-system has overall responsibility for control and starts and stops other sub-systems. It may also devolve control to another subsystembut will expect to have this control responsibility returned to it.
- Event-based control Rather than control information being embedded in a subsystem,each sub-system can respond to externally generated events. These events might come from other sub-systems or from the environment of the system.

# Reference

- Ian Sommerville 2000 Software Engineering. Chapter 11 8th edition