What is software? The programs and other operating information used by a computer.

Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable, and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What are the fundamental software engineering activities?	Software specification, software development, software validation, and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software, and process engineering. Software engineering is part of this more general process.
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times, and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs; 40% are testing costs. For custom software, evolution costs often exceed development costs.
What are the best software engineering techniques and methods?	While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. You can't, therefore, say that one method is better than another.
What differences has the Web made to software engineering?	The Web has led to the availability of software services and the possibility of developing highly distributed service-based systems. Web-based systems development has led to important advances in programming languages and software reuse.

Product characteristics	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security, and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable, and compatible with other systems that they use.

Software Engineering Ethics;

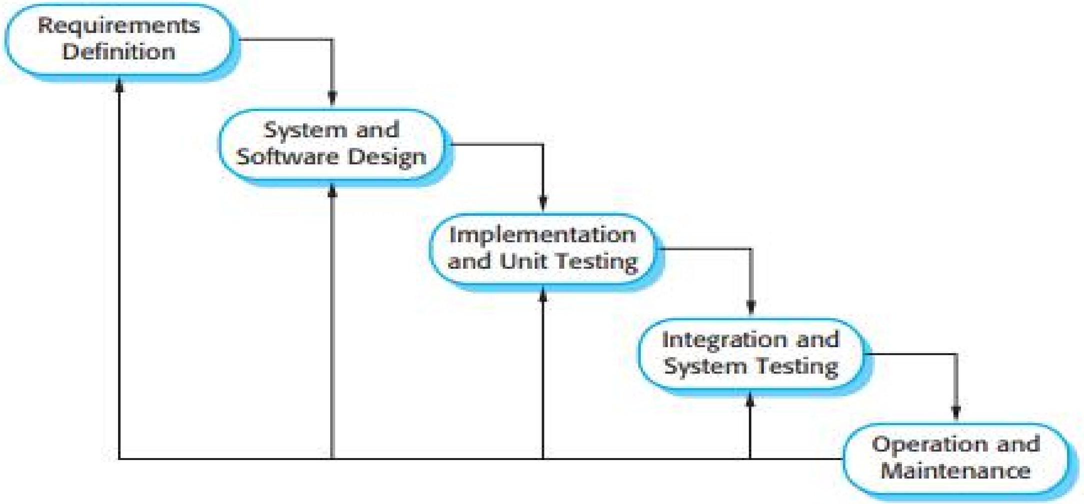
1. *Confidentiality* You should normally respect the confidentiality of your employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.

2. *Competence* You should not misrepresent your level of competence. You should not knowingly accept work that is outside your competence.

3. *Intellectual property rights* You should be aware of local laws governing the use of intellectual property such as patents and copyright. You should be careful to ensure that the intellectual property of employers and clients is protected.

4. *Computer misuse* You should not use your technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses or other malware).

Waterfall Model;



2. *Incremental development* This approach interleaves the activities of specification, development, and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version.

3. *Reuse-oriented software engineering* This approach is based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them from scratch.

1. *Requirements analysis and definition* The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.

2. *System and software design* The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.

3. *Implementation and unit testing* During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

4. *Integration and system testing* The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

5. *Operation and maintenance* Normally (although not necessarily), this is the longest life cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

Software Process Models

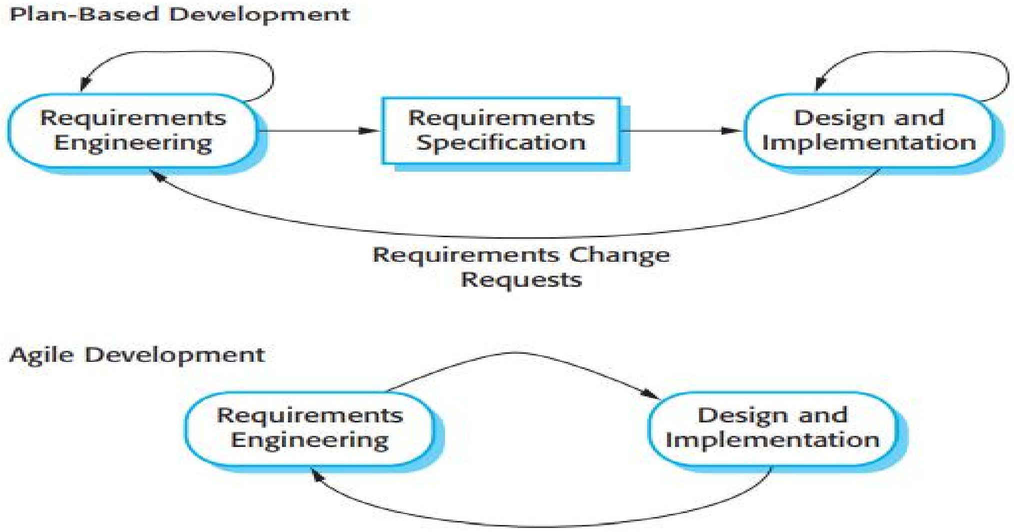
- **The waterfall model**
  - Plan-driven model
  - Separate and distinct phases of specification and development
- **Incremental development**
  - Specification, development and validation are interleaved
  - May be plan-driven or agile
- **Reuse-oriented software engineering**
  - The system is assembled from existing components
  - May be plan-driven or agile

Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models, and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
Testing	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users, and installed in their workplace.
Configuration and change management	This supporting workflow manages changes to the system (see Chapter 25).
Project management	This supporting workflow manages the system development (see Chapters 22 and 23).
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

software quality attributes.

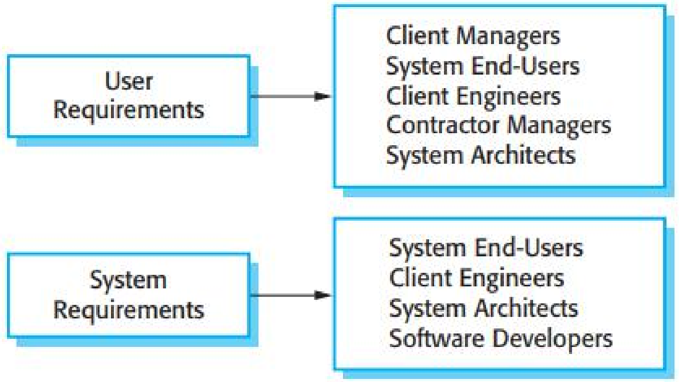
- accessibility
- accountability
- accuracy
- adaptability
- administrability
- affordability
- agility
- auditability
- availability
- credibility
- standards compliance
- process capabilities
- compatibility
- composability
- configurability
- correctness
- customizability
- degradability
- demonstrability
- dependability
- deployability
- distributability
- durability
- evolvability
- extensibility
- fidelity
- flexibility
- installability
- integrity
- interchangeability
- interoperability
- learnability
- maintainability
- manageability
- mobility
- modularity
- nomadicity
- operability
- portability
- precision
- predictability
- recoverability
- relevance
- reliability
- repeatability
- reproducibility
- responsiveness
- reusability
- robustness
- safety
- scalability
- seamlessness
- serviceability (a.k.a supportability)
- securability
- simplicity
- stability
- survivability
- sustainability
- tailorability
- testability
- timeliness
- understandability
- usability

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.



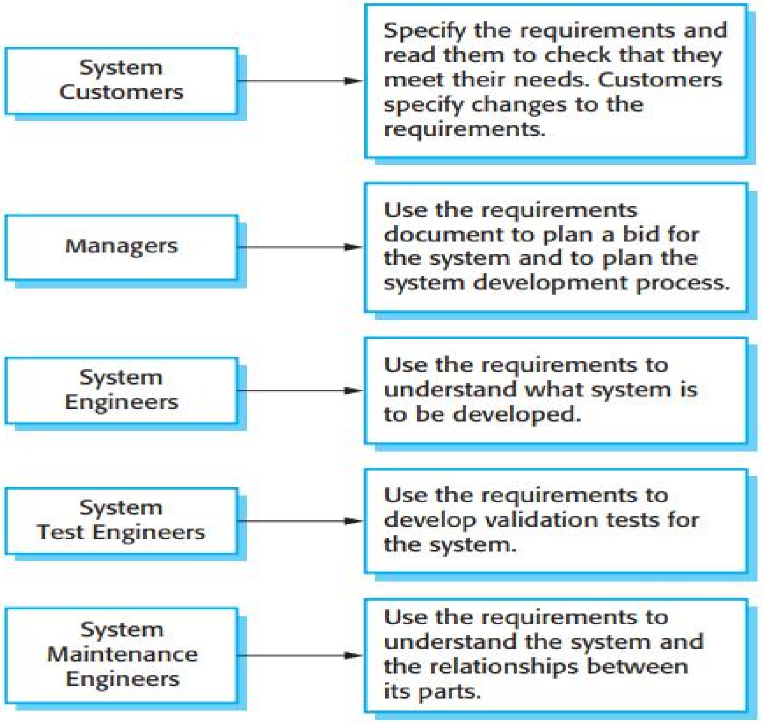
Requirement Analysis;

1. *Functional requirements* These are statements of services the system should provide, how the system should react to particular inputs, and how the system



should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.

2. *Non-functional requirements* These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole, rather than individual system features or services.



Notation	Description
Natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract.

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The non-functional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components, the system, and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Checklist for requirements validation

- Validity. Does the system provide the functions which best support the customer's needs?
- Consistency. Are there any requirements conflicts?
- Completeness. Are all functions required by the customer included?
- Realism. Can the requirements be implemented given available budget and technology?
- Verifiability. Can the requirements be verified (e.g. tested)?

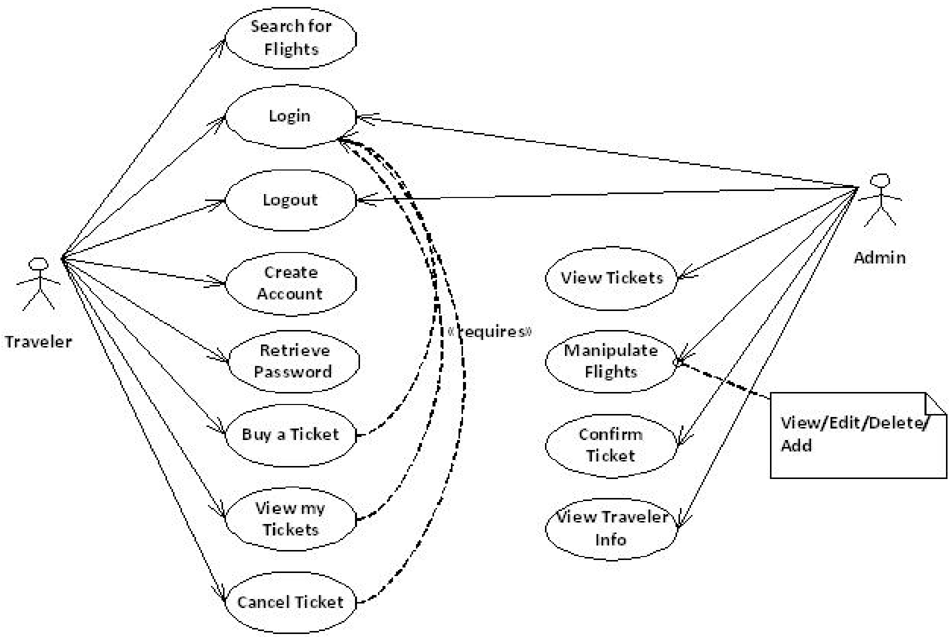
Requirements validation techniques

- Requirements review
  - Systematic manual analysis of the requirements.
- Prototyping
  - Using an executable model of the system to check requirements (see Chapter 2).
- Test-case generation
  - Developing tests for requirements to check testability.

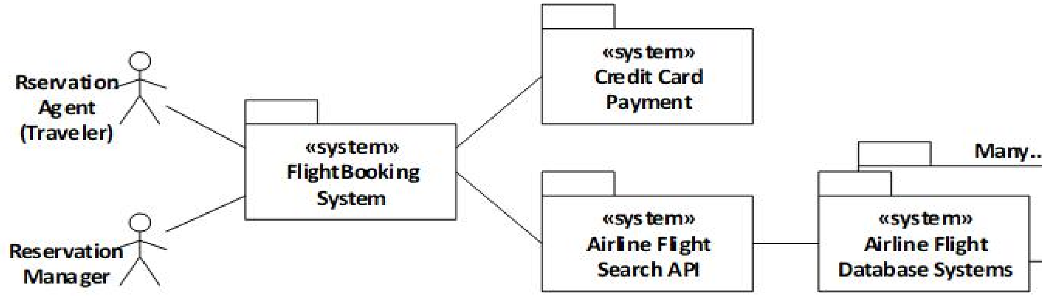
Checklist for requirements review

- Verifiability
  - Is the requirement realistically testable?
- Comprehensibility
  - Is the requirement properly understood?
- Traceability
  - Is the origin of the requirement clearly stated?
- Adaptability
  - Can the requirement be changed without a large impact on other requirements?

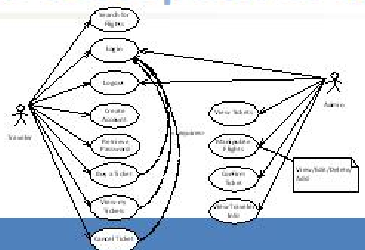
FBS – Use-Case Diagram



FBS – Context Diagram

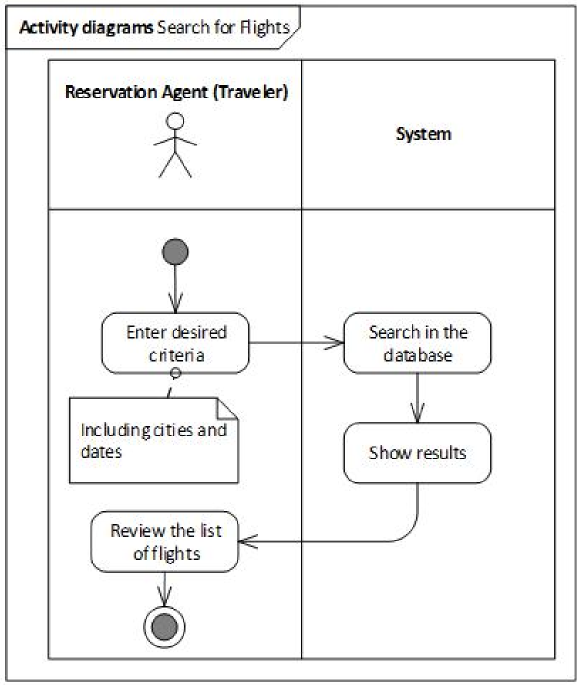


Tabular description of use-cases



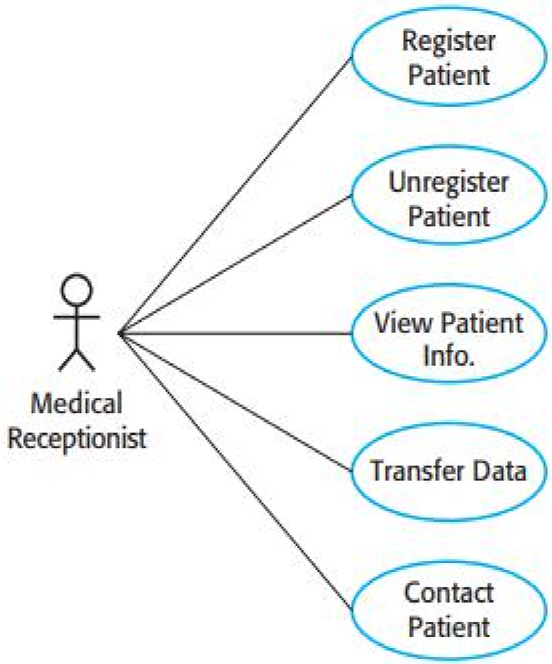
Use case: Login	
Actors	Traveler
Precondition	Traveler is not logged in
Post-condition	Traveler is logged in
Main (happy) path:	1. User enters her/his username and password and presses on the Login button 2. System checks the username and password and shows the app's main window (page) 3. User will be able to use the feature afterwards
Alternative path:	1. If the username and password combination is invalid, the system will show a message indicating it 2. User will be able to enter another username and password

FBS – Details of "Search for Flights" use-case desc. by Activity Diagram

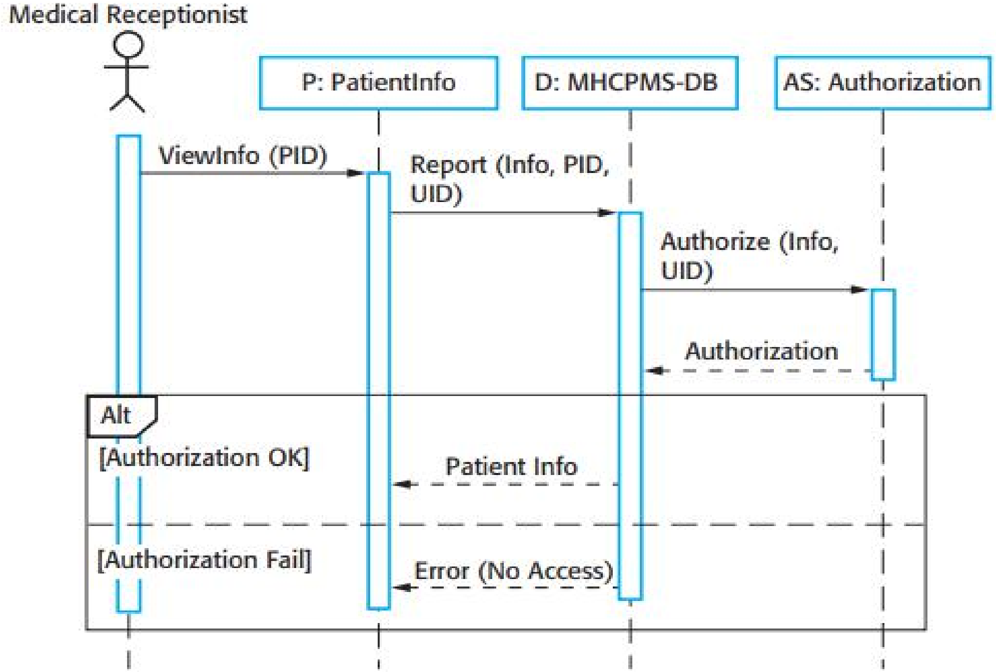


Use Case Diagram;

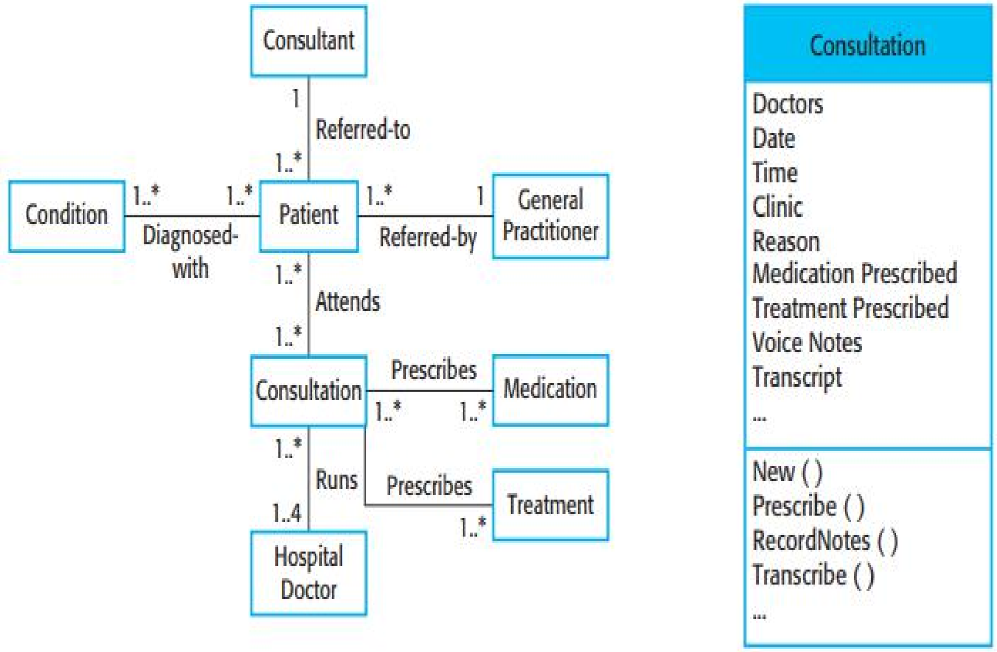
MHC-PMS: Transfer data	
Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the MHC-PMS to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.



Sequence Diagram;

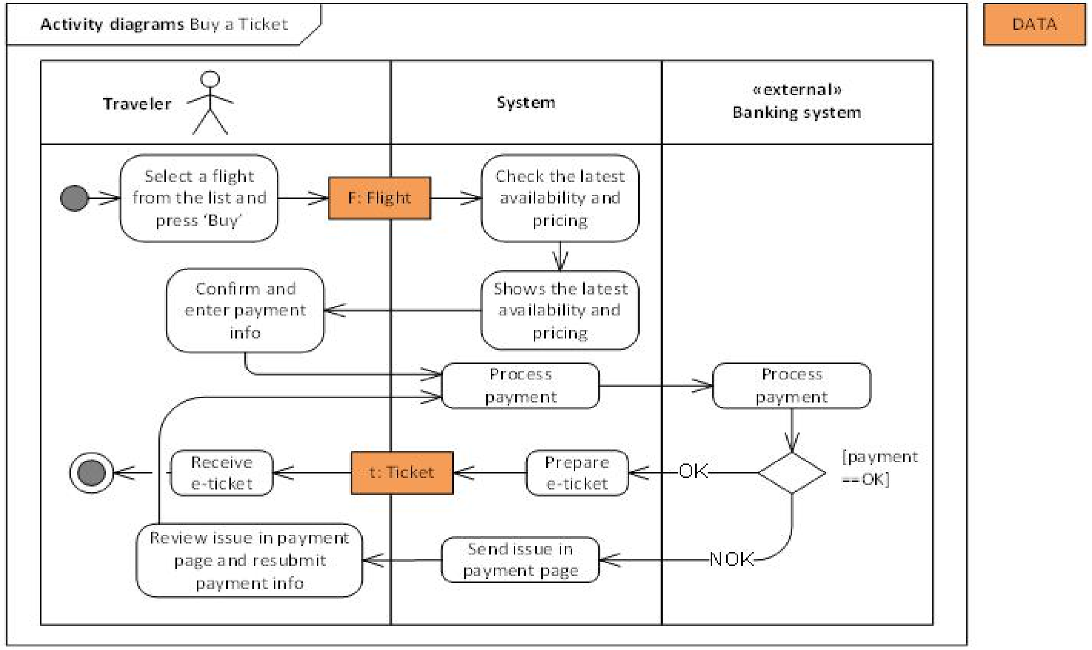


Class Diagram;

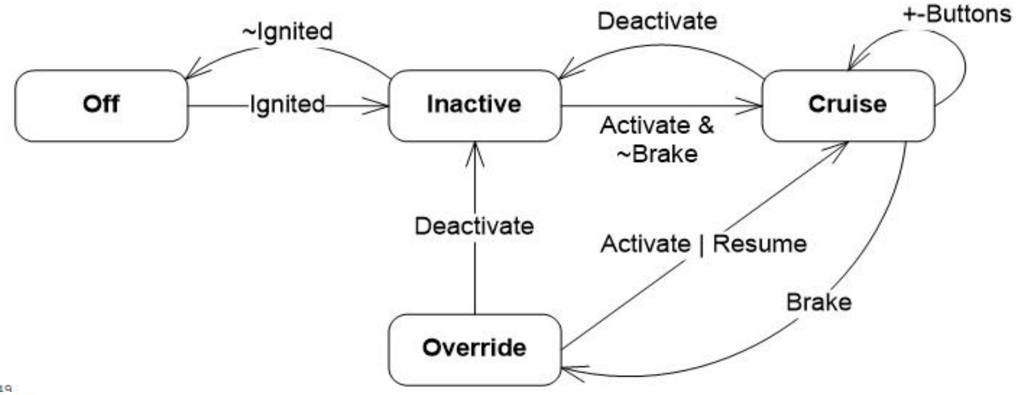


Architectural Design;

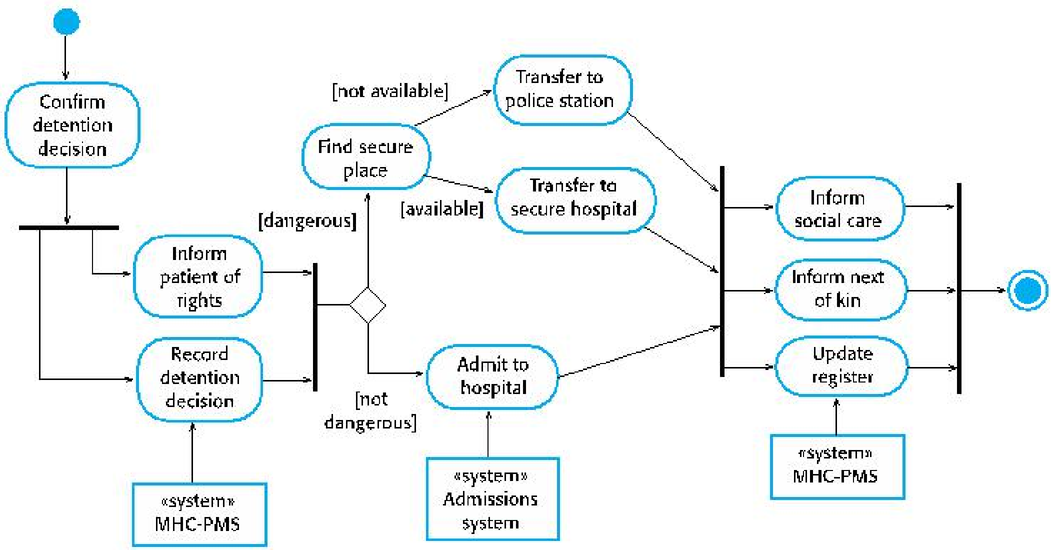
Data-driven modeling using activity diagrams



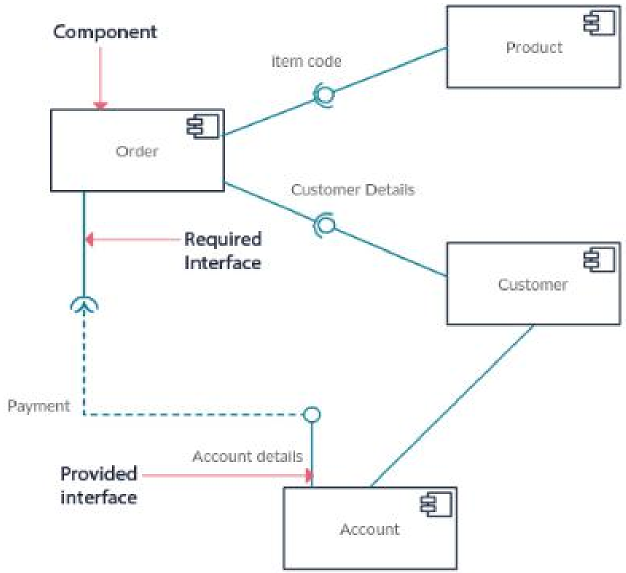
State charts a cruise control system



Process model of involuntary detention

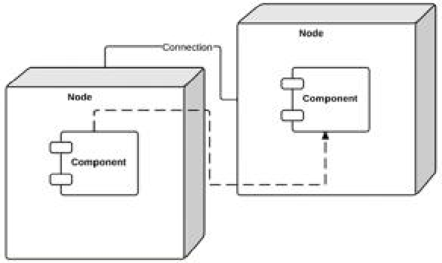


UML Component Diagram - Example

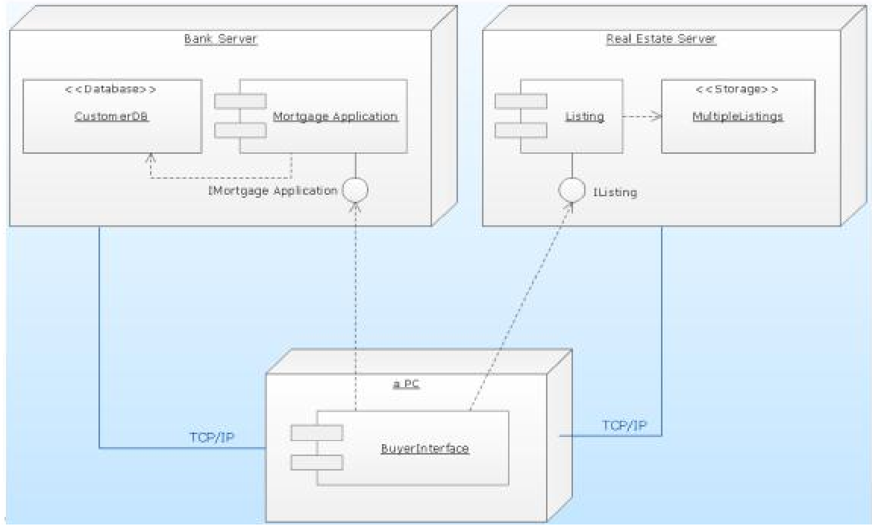


https://www.youtube.com/watch?v=eI5RA45OQGs&list=PLh9ECzBB8tJOmOydGaV7xr9h13ooEjBkV&index=7
https://www.youtube.com/watch?v=AI7ngIoIHeQ&list=PLh9ECzBB8tJOmOydGaV7xr9h13ooEjBkV&index=8

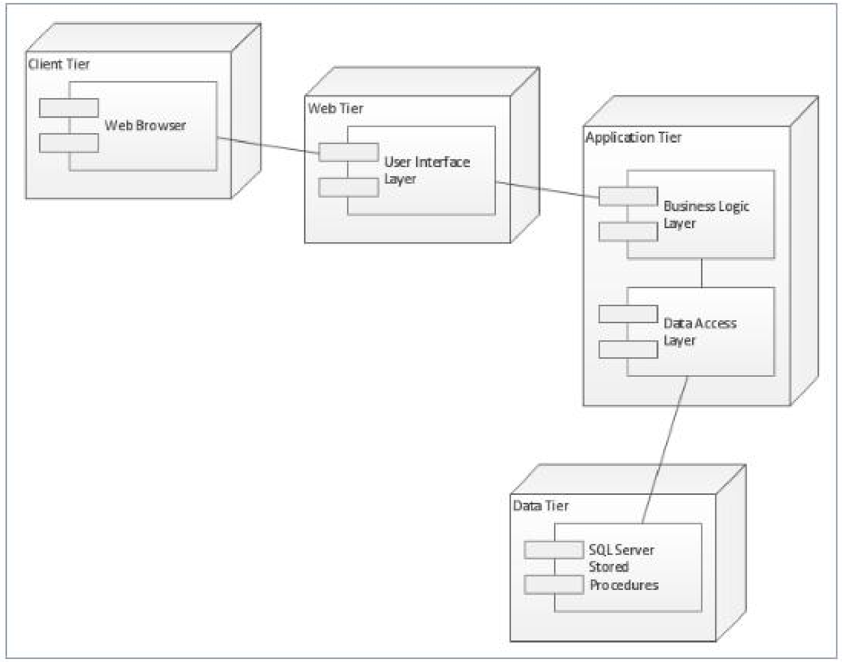
UML Deployment Diagram: Example



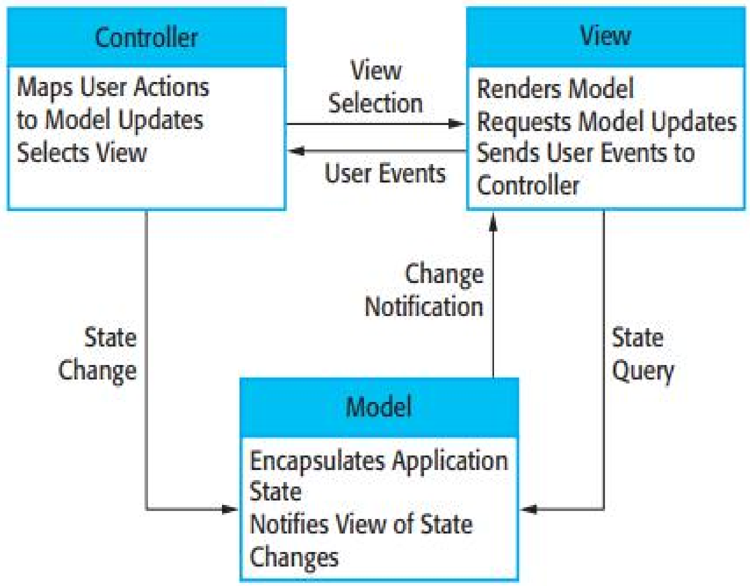
http://www.conceptdraw.com/How-To-Guide/uml-deployment-diagram



UML Deployment Diagram: 4-Tier Architecture

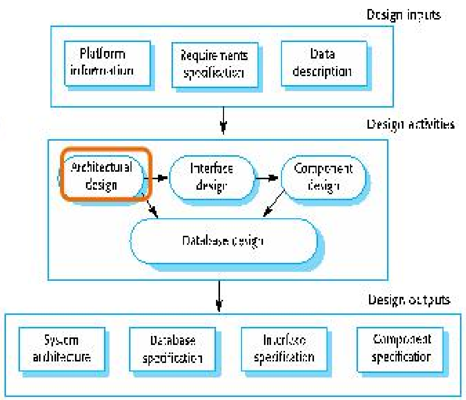


Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines 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. See Figure 6.3.
Example	Figure 6.4 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. 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.



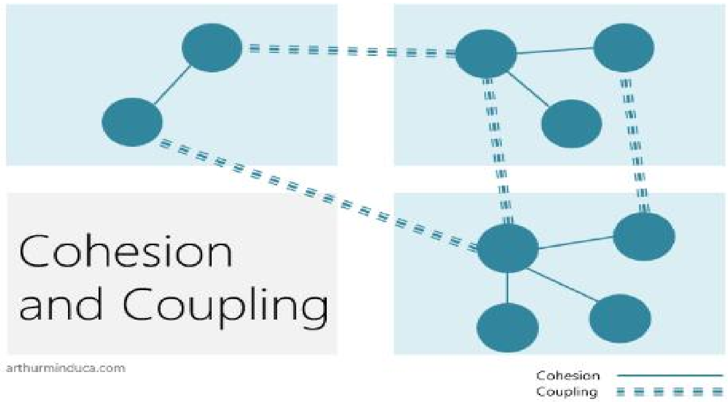
Design activities (Ch.2)

- *Architectural design*, where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships and how they are distributed.
- *Interface design*, interfaces between system components.
- *Component design*, you take each system component and design how it will operate.
- *Database design*, you design the system data structures and how these are to be represented in a database.



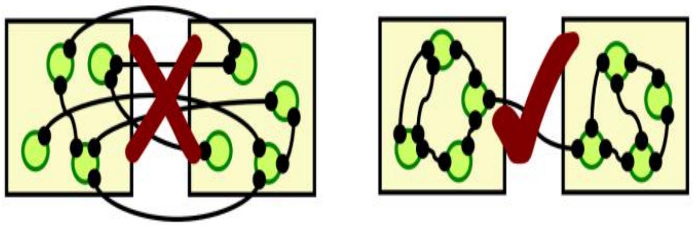
An important topic in software architecture: Cohesion and Coupling

- Cohesion is the degree to which the elements in a design unit (package, class etc.) are logically related, or "belong together".
- Coupling is the degree to which the elements in a design are connected.

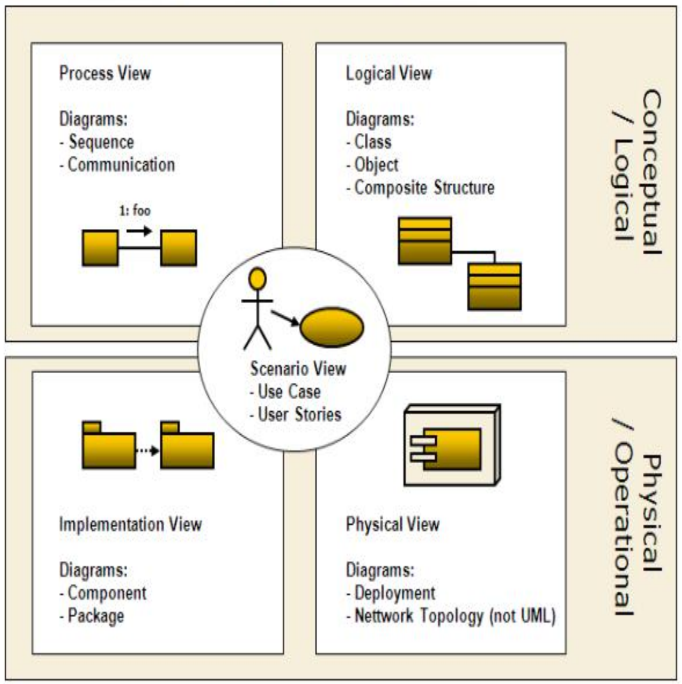


Cohesion and Coupling

- A good architecture:
  - Maximizes the **cohesion** of each module
    - Goal: the contents of each module are strongly inter-related
    - High cohesion means the subcomponents really do belong together
  - Minimizes **coupling** between modules:
    - Goal: modules don't need to know much about one another to interact
    - Low coupling makes future change easier



4 + 1 view model of software architecture (introduced by Philippe Kruchten)



https://www.researchgate.net/publication/238381956_A_41_View_Model_of_Software_Architecture

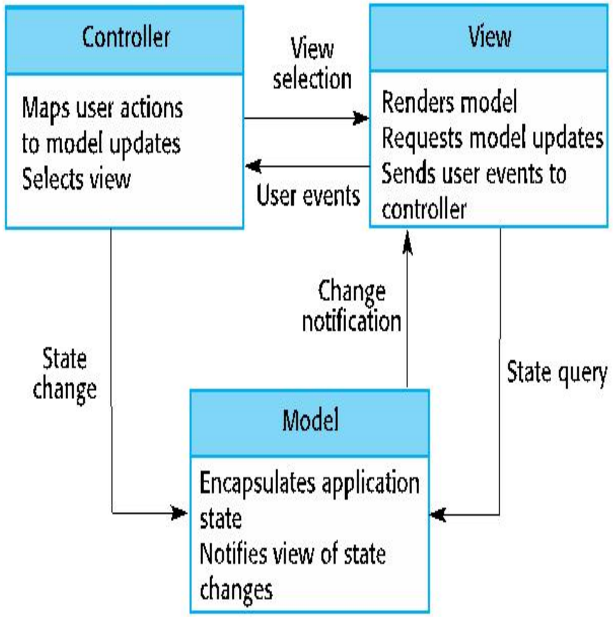
4 + 1 view model of software architecture

For both design and documentation, you usually need to present multiple views of the software architecture:

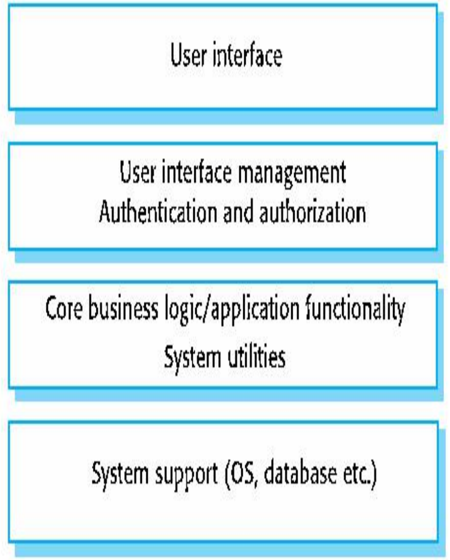
- A logical view, which shows the key abstractions in the system as objects or object classes.
- A process view, which shows how, at run-time, the system is composed of interacting processes.
- A development view, which shows how the software is decomposed for development.
- A physical view, which shows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

Architectural Pattern;

The organization of the Model-View-Controller

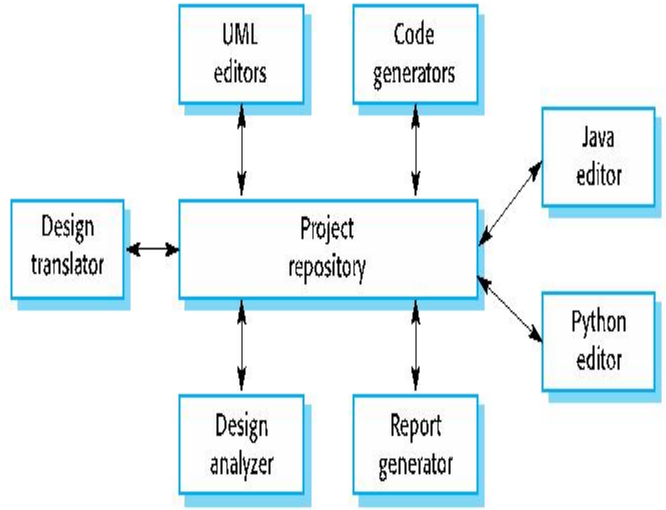


2 - Layered architecture pattern

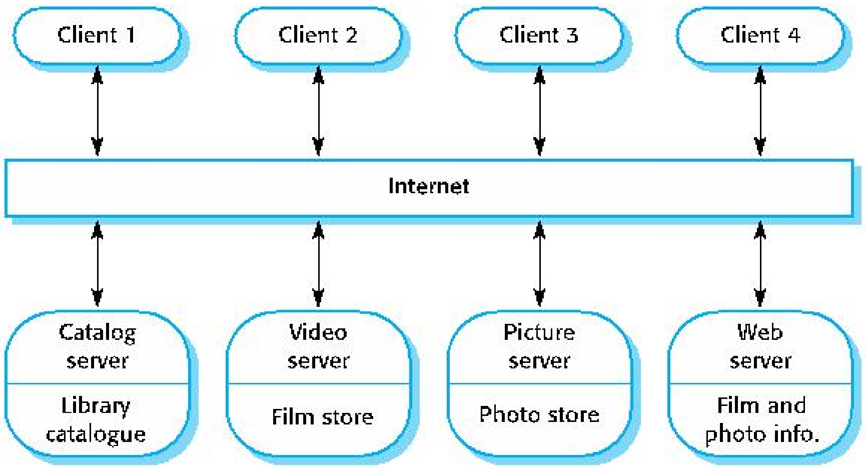


3 - Repository architecture pattern

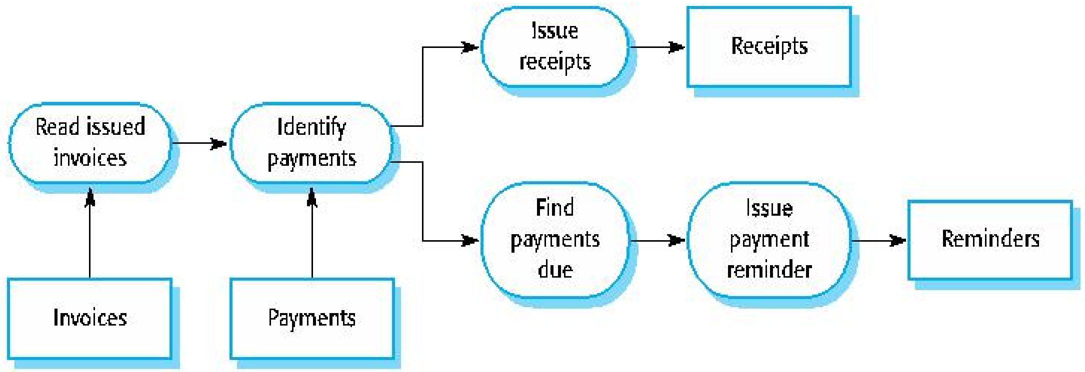
A repository architecture for an IDE



A client–server architecture for a film library



An example of the pipe and filter architecture



Risk Management;

We need management activities!

- Project planning
  - Project managers are responsible for planning. estimating and scheduling project development and assigning people to tasks.
- Reporting
  - Project managers are usually responsible for reporting on the progress of a project to customers and to the managers of the company developing the software.
- Risk management
  - Project managers assess the risks that may affect a project, monitor these risks and take action when problems arise.
- People management
  - Project managers have to choose people for their team and establish ways of working that leads to effective team performance
- Proposal writing
  - The first stage in a software project may involve writing a proposal to win a contract to carry out an item of work. The proposal describes the objectives of the project and how it will be carried out.

Risk management

- Risk management is concerned with identifying risks and drawing up plans to minimise their effect on a project.

- A risk is a probability that some adverse circumstance will occur
  - Project risks affect schedule or resources;
  - Product risks affect the quality or performance of the software being developed;
  - Business risks affect the organisation developing or procuring the software.

The risk management process

- Risk identification - Identify project, product and business risks;
- Risk analysis - Assess the likelihood and consequences of these risks;
- Risk planning - Draw up plans to avoid or minimise effects of the risk;
- Risk monitoring - Monitor the risks throughout the project.