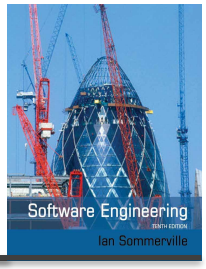
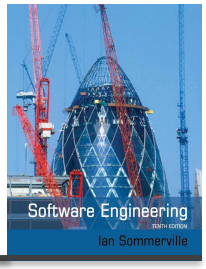


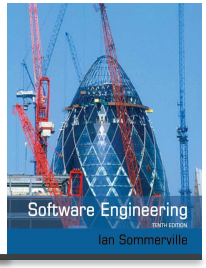
Chapter 4 – Requirements Engineering

Topics covered



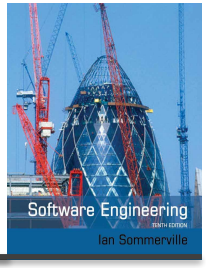
- Functional and non-functional requirements
- Requirements engineering processes
- Requirements elicitation
- Requirements specification
- Requirements validation
- Requirements change

Requirements engineering



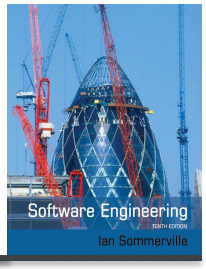
- The process of establishing the services that acustomer requires from a system and the constraints under which it operates and is developed.
- The system requirements are the descriptions of the system services and constraints that are generated during the requirements engineering process.

What is a requirement?



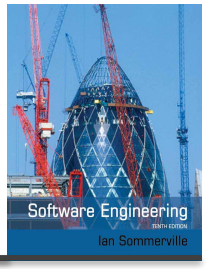
- It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- This is inevitable as requirements may serve a dual function
  - May be the basis for a bid for a contract - therefore must be open to interpretation;
  - May be the basis for the contract itself - therefore must be defined in detail;
  - Both these statements may be called requirements.

Requirements abstraction (Davis)



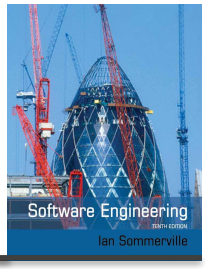
"If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization's needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system."

Types of requirement



- User requirements
  - Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.
- System requirements
  - A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

User and system requirements



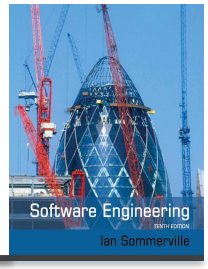
User requirements definition

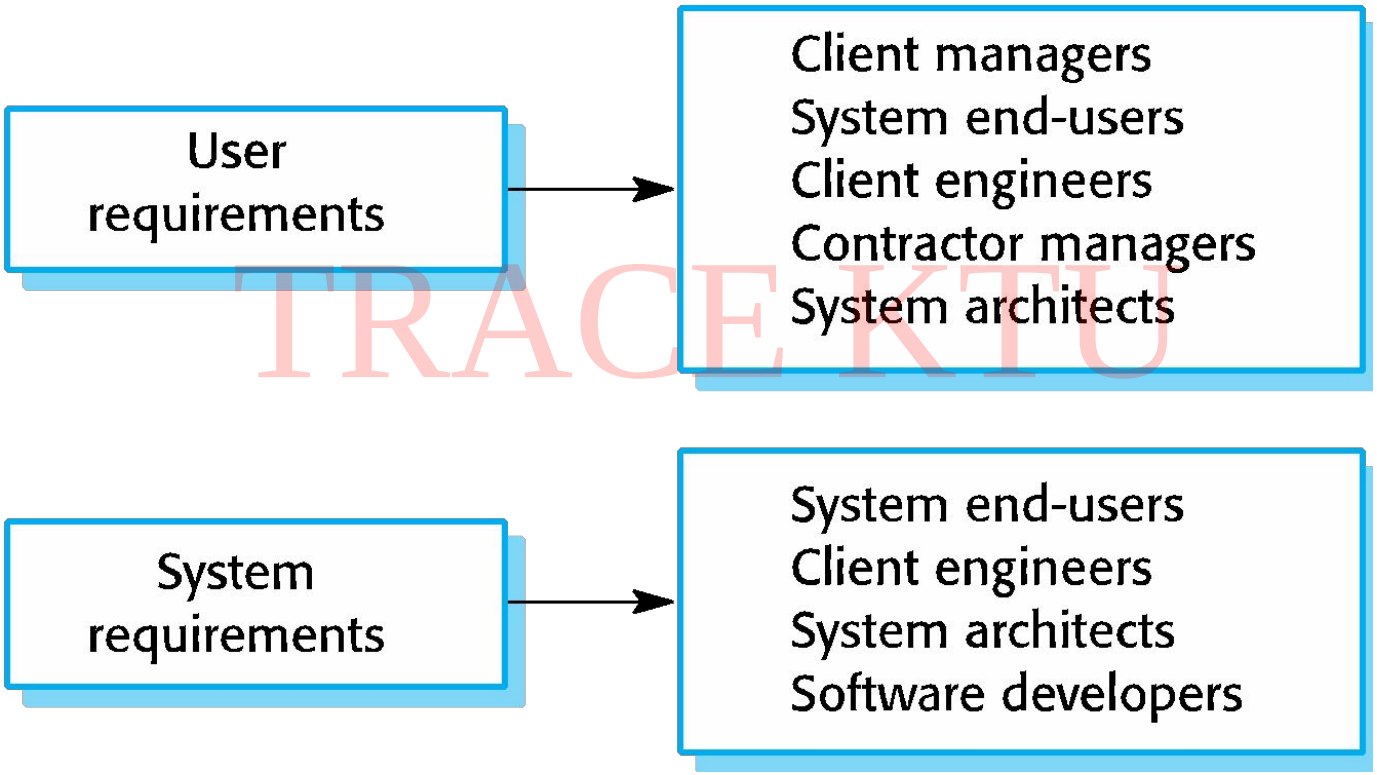
1. The Mentcare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System requirements specification

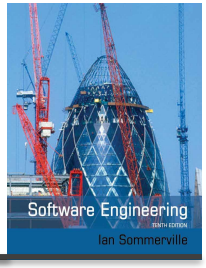
1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
1.2 The system shall generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g. 10mg, 20mg, etc) separate reports shall be created for each dose unit.
1.5 Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

Readers of different types of requirements specification



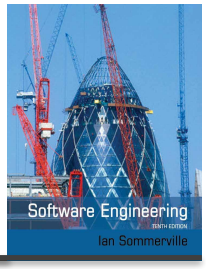


System stakeholders



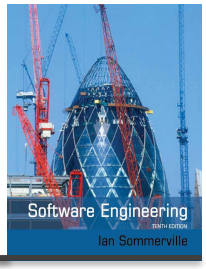
- Any person or organization who is affected by the system in some way and so who has a legitimate interest
- Stakeholder types
  - End users
  - System managers
  - System owners
  - External stakeholders

Stakeholders in the Mentcare system



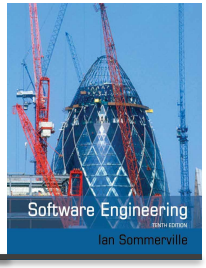
- Patients whose information is recorded in the system.
- Doctors who are responsible for assessing and treating patients.
- Nurses who coordinate the consultations with doctors and administer some treatments.
- Medical receptionists who manage patients' appointments.
- IT staff who are responsible for installing and maintaining the system.

Stakeholders in the Mentcare system

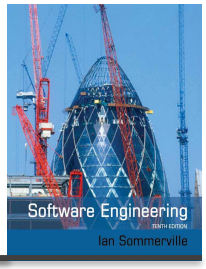


- A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
- Health care managers who obtain management information from the system.
- Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

Agile methods and requirements

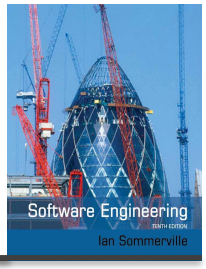


- Many agile methods argue that producing detailed system requirements is a waste of time as requirements change so quickly.
- The requirements document is therefore always out of date.
- Agile methods usually use incremental requirements engineering and may express requirements as ‘user stories’ (discussed in Chapter 3).
- This is practical for business systems but problematic for systems that require pre-delivery analysis (e.g. critical systems) or systems developed by several teams.



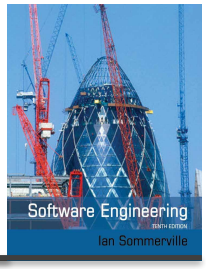
Functional and non-functional requirements

Functional and non-functional requirements



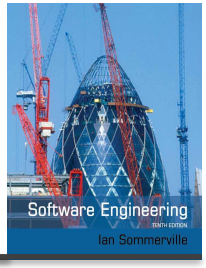
- Functional requirements
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
  - May state what the system should not do.
- Non-functional requirements
  - Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
  - Often apply to the system as a whole rather than individual features or services.
- Domain requirements
  - Constraints on the system from the domain of operation

Functional requirements



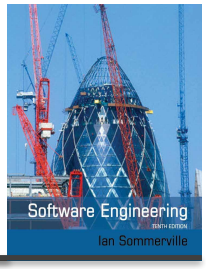
- Describe functionality or system services.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do.
- Functional system requirements should describe the system services in detail.

Mentcare system: functional requirements



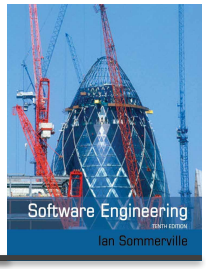
- A user shall be able to search the appointments lists for all clinics.
- The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
- Each staff member using the system shall be uniquely identified by his or her 8-digit employee number.

Requirements imprecision



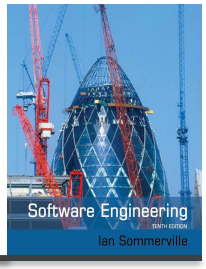
- Problems arise when functional requirements are not precisely stated.
- Ambiguous requirements may be interpreted in different ways by developers and users.
- Consider the term ‘search’ in requirement 1
  - User intention – search for a patient name across all appointments in all clinics;
  - Developer interpretation – search for a patient name in an individual clinic. User chooses clinic then search.

Requirements completeness and consistency



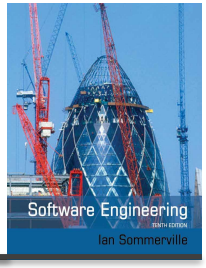
- In principle, requirements should be both complete and consistent.
- Complete
  - They should include descriptions of all facilities required.
- Consistent
  - There should be no conflicts or contradictions in the descriptions of the system facilities.
- In practice, because of system and environmental complexity, it is impossible to produce a complete and consistent requirements document.

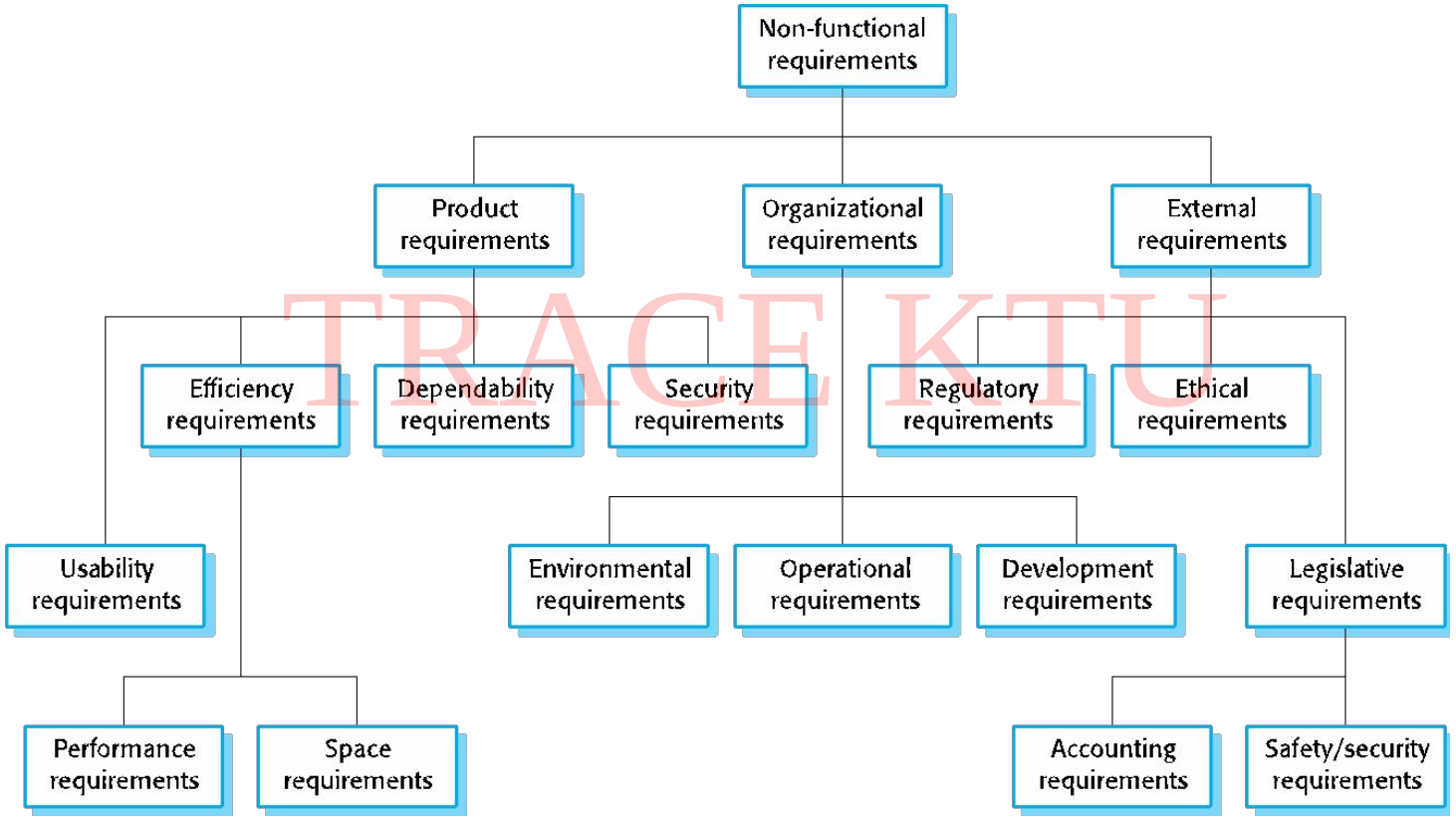
Non-functional requirements



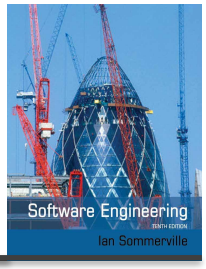
- These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular IDE, programming language or development method.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system may be useless.

Types of nonfunctional requirement



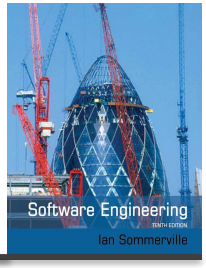


Non-functional requirements implementation



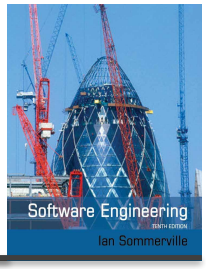
- Non-functional requirements may affect the overall architecture of a system rather than the individual components.
  - For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
- A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define system services that are required.
  - It may also generate requirements that restrict existing requirements.

Non-functional classifications



- Product requirements
  - Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- Organisational requirements
  - Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- External requirements
  - Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Examples of nonfunctional requirements in the Mentcare system

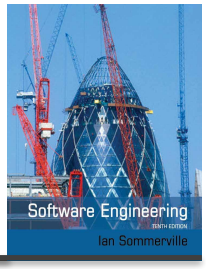


Product requirement
The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 0830–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

Organizational requirement
Users of the Mentcare system shall authenticate themselves using their health authority identity card.

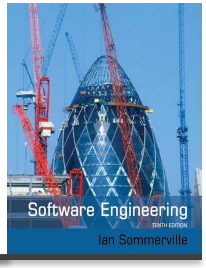
External requirement
The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Goals and requirements



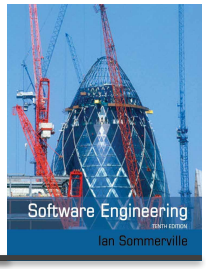
- Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.
- Goal
  - A general intention of the user such as ease of use.
- Verifiable non-functional requirement
  - A statement using some measure that can be objectively tested.
- Goals are helpful to developers as they convey the intentions of the system users.

Usability requirements

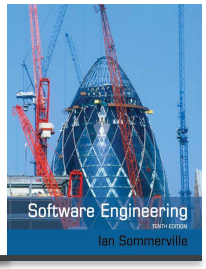


- The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized. (Goal)
- Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use. (Testable non-functional requirement)

Metrics for specifying nonfunctional requirements

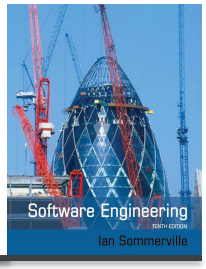


Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems



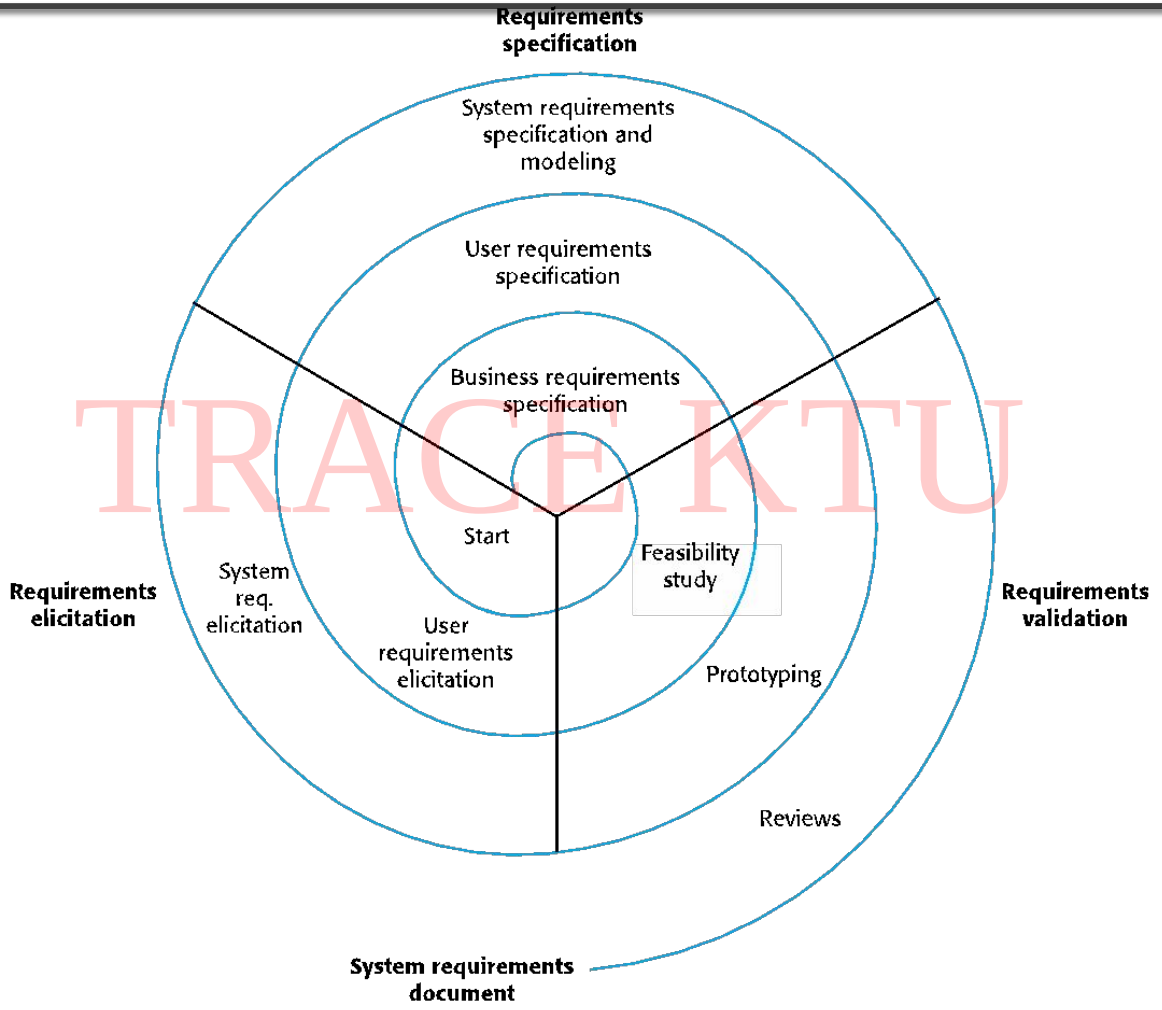
Requirements engineering processes

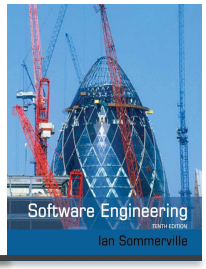
Requirements engineering processes



- The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements.
- However, there are a number of generic activities common to all processes
  - Requirements elicitation;
  - Requirements analysis;
  - Requirements validation;
  - Requirements management.
- In practice, RE is an iterative activity in which these processes are interleaved.

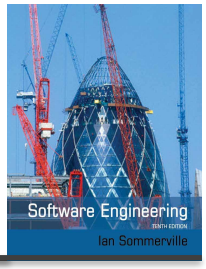
A spiral view of the requirements engineering process



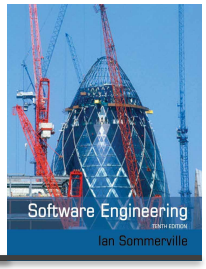


Requirements elicitation

Requirements elicitation and analysis

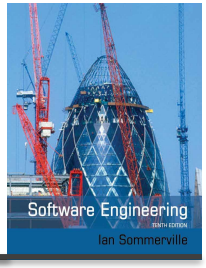


- Sometimes called requirements elicitation or requirements discovery.
- Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders.*



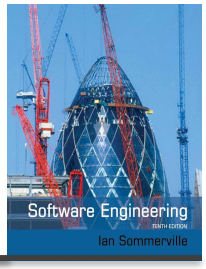
Requirements elicitation

Requirements elicitation



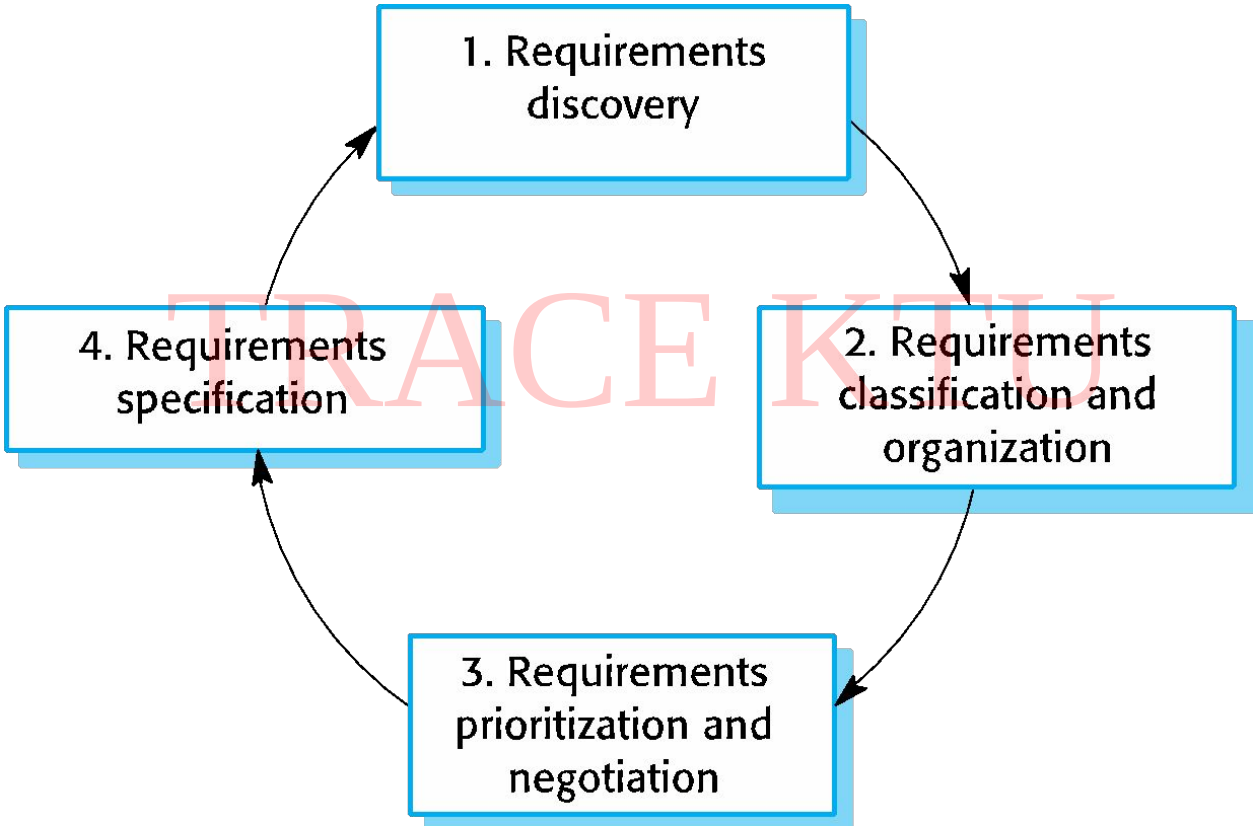
- Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide, the required system performance, hardware constraints, other systems, etc.
- Stages include:
  - Requirements discovery,
  - Requirements classification and organization,
  - Requirements prioritization and negotiation,
  - Requirements specification.

Problems of requirements elicitation

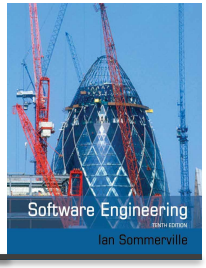


- Stakeholders don't know what they really want.
- Stakeholders express requirements in their own terms.
- Different stakeholders may have conflicting requirements.
- Organisational and political factors may influence the system requirements.
- The requirements change during the analysis process. New stakeholders may emerge and the business environment may change.

The requirements elicitation and analysis process

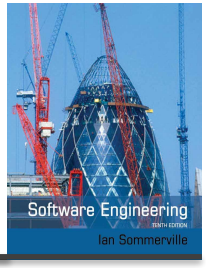


Process activities



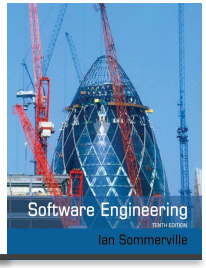
- Requirements discovery
  - Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.
- Requirements classification and organisation
  - Groups related requirements and organises them into coherent clusters.
- Prioritisation and negotiation
  - Prioritising requirements and resolving requirements conflicts.
- Requirements specification
  - Requirements are documented and input into the next round of the spiral.

Requirements discovery



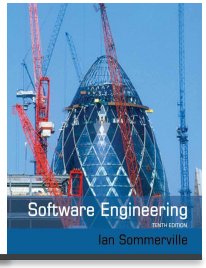
- The process of gathering information about the required and existing systems and distilling the user and system requirements from this information.
- Interaction is with system stakeholders from managers to external regulators.
- Systems normally have a range of stakeholders.

Interviewing



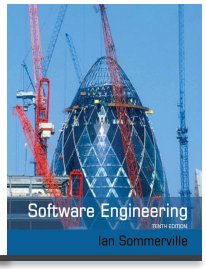
- Formal or informal interviews with stakeholders are part of most RE processes.
- Types of interview
  - Closed interviews based on pre-determined list of questions
  - Open interviews where various issues are explored with stakeholders.
- Effective interviewing
  - Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
  - Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.

Interviews in practice



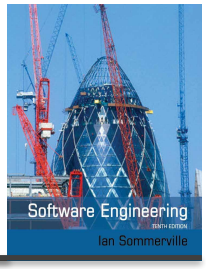
- Normally a mix of closed and open-ended interviewing.
- Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.
- Interviewers need to be open-minded without pre-conceived ideas of what the system should do
- You need to prompt the use to talk about the system by suggesting requirements rather than simply asking them what they want.

Problems with interviews



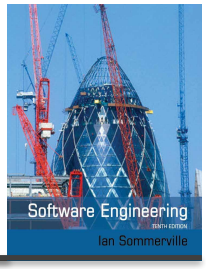
- Application specialists may use language to describe their work that isn't easy for the requirements engineer to understand.
- Interviews are not good for understanding domain requirements
  - Requirements engineers cannot understand specific domain terminology;
  - Some domain knowledge is so familiar that people find it hard to articulate or think that it isn't worth articulating.

Ethnography



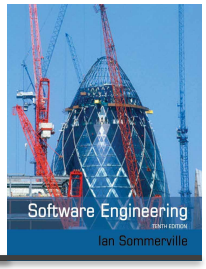
- A social scientist spends a considerable time observing and analysing how people actually work.
- People do not have to explain or articulate their work.
- Social and organisational factors of importance may be observed.
- Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.

Scope of ethnography



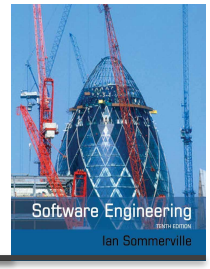
- Requirements that are derived from the way that people actually work rather than the way I which process definitions suggest that they ought to work.
- Requirements that are derived from cooperation and awareness of other people's activities.
  - Awareness of what other people are doing leads to changes in the ways in which we do things.
- Ethnography is effective for understanding existing processes but cannot identify new features that should be added to a system.

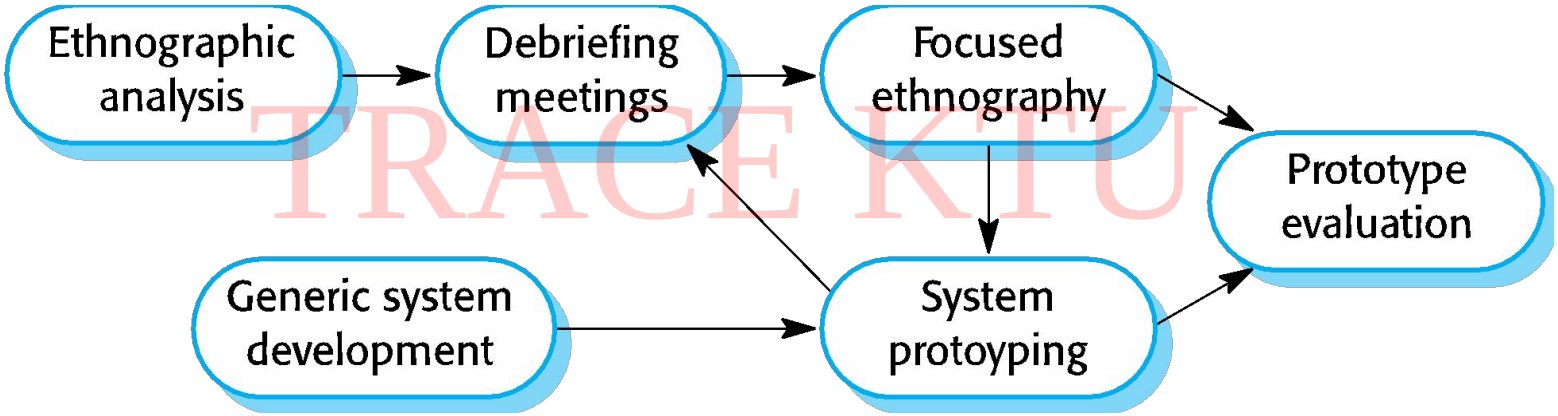
Focused ethnography



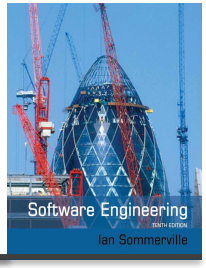
- Developed in a project studying the air traffic control process
- Combines ethnography with prototyping
- Prototype development results in unanswered questions which focus the ethnographic analysis.
- The problem with ethnography is that it studies existing practices which may have some historical basis which is no longer relevant.

Ethnography and prototyping for requirements analysis



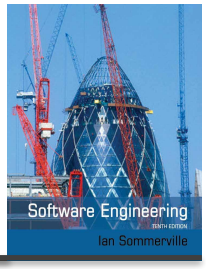


Stories and scenarios

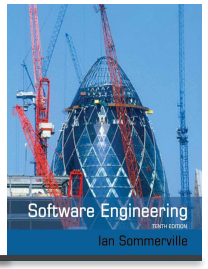


- Scenarios and user stories are real-life examples of how a system can be used.
- Stories and scenarios are a description of how a system may be used for a particular task.
- Because they are based on a practical situation, stakeholders can relate to them and can comment on their situation with respect to the story.

Scenarios



- A structured form of user story
- Scenarios should include
  - A description of the starting situation;
  - A description of the normal flow of events;
  - A description of what can go wrong;
  - Information about other concurrent activities;
  - A description of the state when the scenario finishes.

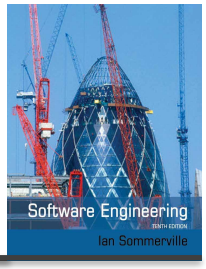


Scenario for collecting medical history in MHC-PMS

Initial assumption: The patient has seen a medical receptionist who has created a record in the system and collected the patient's personal information (name, address, age, etc.). A nurse is logged on to the system and is collecting medical history.

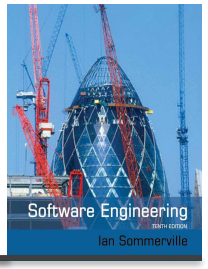
Normal: The nurse searches for the patient by family name. If there is more than one patient with the same surname, the given name (first name in English) and date of birth are used to identify the patient.

The nurse chooses the menu option to add medical history.



The nurse chooses the menu option to add medical history.

The nurse then follows a series of prompts from the system to enter information about consultations elsewhere on mental health problems (free text input), existing medical conditions (nurse selects conditions from menu), medication currently taken (selected from menu), allergies (free text), and home life (form).



Scenario for collecting medical history in MHC-PMS (Conti.

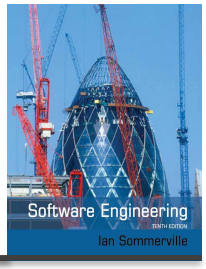
What can go wrong: The patient's record does not exist or cannot be found. The nurse should create a new record and record personal information.

Patient conditions or medication are not entered in the menu. The nurse should choose the 'other' option and enter free text describing the condition/medication.

Patient cannot/will not provide information on medical history. The nurse should enter free text recording the patient's inability/unwillingness to provide information. The system should print the standard exclusion form stating that the lack of information may mean that treatment will be limited or delayed. This should be signed and handed to the patient.

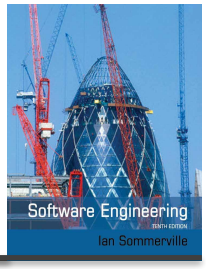
Other activities: Record may be consulted but not edited by other staff while information is being entered.

System state on completion: User is logged on. The patient record including medical history is entered in the database, a record is added to the system log showing the start and end time of the session and the nurse involved.



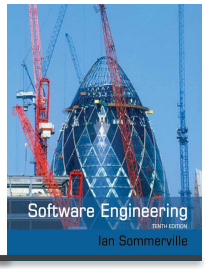
Requirements specification

Requirements specification



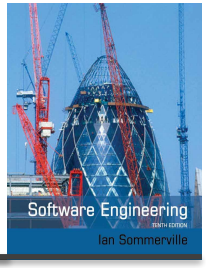
- The process of writing the user and system requirements in a requirements document.
- User requirements have to be understandable by end-users and customers who do not have a technical background.
- System requirements are more detailed requirements and may include more technical information.
- The requirements may be part of a contract for the system development
  - It is therefore important that these are as complete as possible.

Ways of writing a system requirements specification



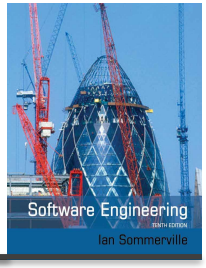
Notation	Description
Natural language	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about 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

Requirements and design



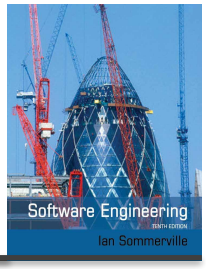
- In principle, requirements should state what the system should do and the design should describe how it does this.
- In practice, requirements and design are inseparable
  - A system architecture may be designed to structure the requirements;
  - The system may inter-operate with other systems that generate design requirements;
  - The use of a specific architecture to satisfy non-functional requirements may be a domain requirement.
  - This may be the consequence of a regulatory requirement.

Natural language specification



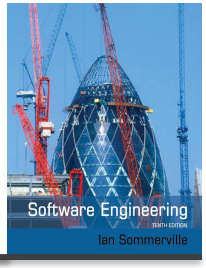
- Requirements are written as natural language sentences supplemented by diagrams and tables.
- Used for writing requirements because it is expressive, intuitive and universal. This means that the requirements can be understood by users and customers.

Guidelines for writing requirements



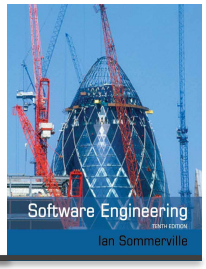
- Invent a standard format and use it for all requirements.
- Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
- Use text highlighting to identify key parts of the requirement.
- Avoid the use of computer jargon.
- Include an explanation (rationale) of why a requirement is necessary.

Problems with natural language



- Lack of clarity
  - Precision is difficult without making the document difficult to read.
- Requirements confusion
  - Functional and non-functional requirements tend to be mixed-up.
- Requirements amalgamation
  - Several different requirements may be expressed together.

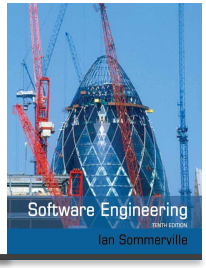
Example requirements for the insulin pump software system



3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. *(Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.)*

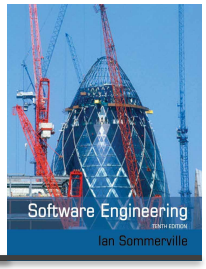
3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. *(A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.)*

Structured specifications



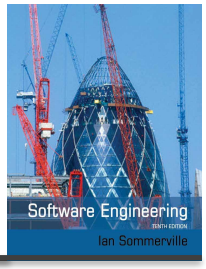
- An approach to writing requirements where the freedom of the requirements writer is limited and requirements are written in a standard way.
- This works well for some types of requirements e.g. requirements for embedded control system but is sometimes too rigid for writing business system requirements.

Form-based specifications



- Definition of the function or entity.
- Description of inputs and where they come from.
- Description of outputs and where they go to.
- Information about the information needed for the computation and other entities used.
- Description of the action to be taken.
- Pre and post conditions (if appropriate).
- The side effects (if any) of the function.

A structured specification of a requirement for an insulin pump



Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: safe sugar level.

Description

Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.

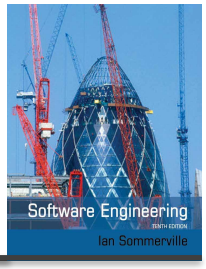
Inputs Current sugar reading (r2); the previous two readings (r0 and r1).

Source Current sugar reading from sensor. Other readings from memory.

Outputs CompDose—the dose in insulin to be delivered.

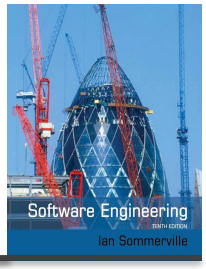
Destination Main control loop.

A structured specification of a requirement for an insulin pump



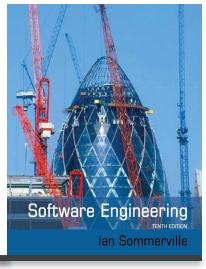
Action
CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.
Requirements
Two previous readings so that the rate of change of sugar level can be computed.
Pre-condition
The insulin reservoir contains at least the maximum allowed single dose of insulin.
Post-condition r0 is replaced by r1 then r1 is replaced by r2.
Side effects None.

Tabular specification



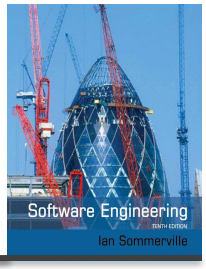
- Used to supplement natural language.
- Particularly useful when you have to define a number of possible alternative courses of action.
- For example, the insulin pump systems bases its computations on the rate of change of blood sugar level and the tabular specification explains how to calculate the insulin requirement for different scenarios.

Tabular specification of computation for an insulin pump

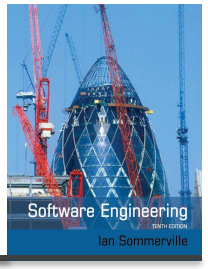


Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 – r1) < (r1 – r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing ((r2 – r1) ≥ (r1 – r0))	CompDose = round ((r2 – r1)/4) If rounded result = 0 then CompDose = MinimumDose

Use cases



- Use-cases are a kind of scenario that are included in the UML.
- Use cases identify the actors in an interaction and which describe the interaction itself.
- A set of use cases should describe all possible interactions with the system.
- High-level graphical model supplemented by more detailed tabular description (see Chapter 5).
- UML sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.

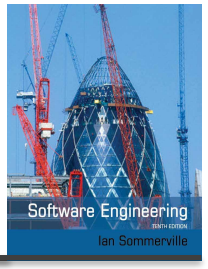


Use case modeling

- Use cases were developed originally to support requirements elicitation and now incorporated into the UML.
- Each use case represents a discrete task that involves external interaction with a system.
- Actors in a use case may be people or other systems.

Transfer-data use case

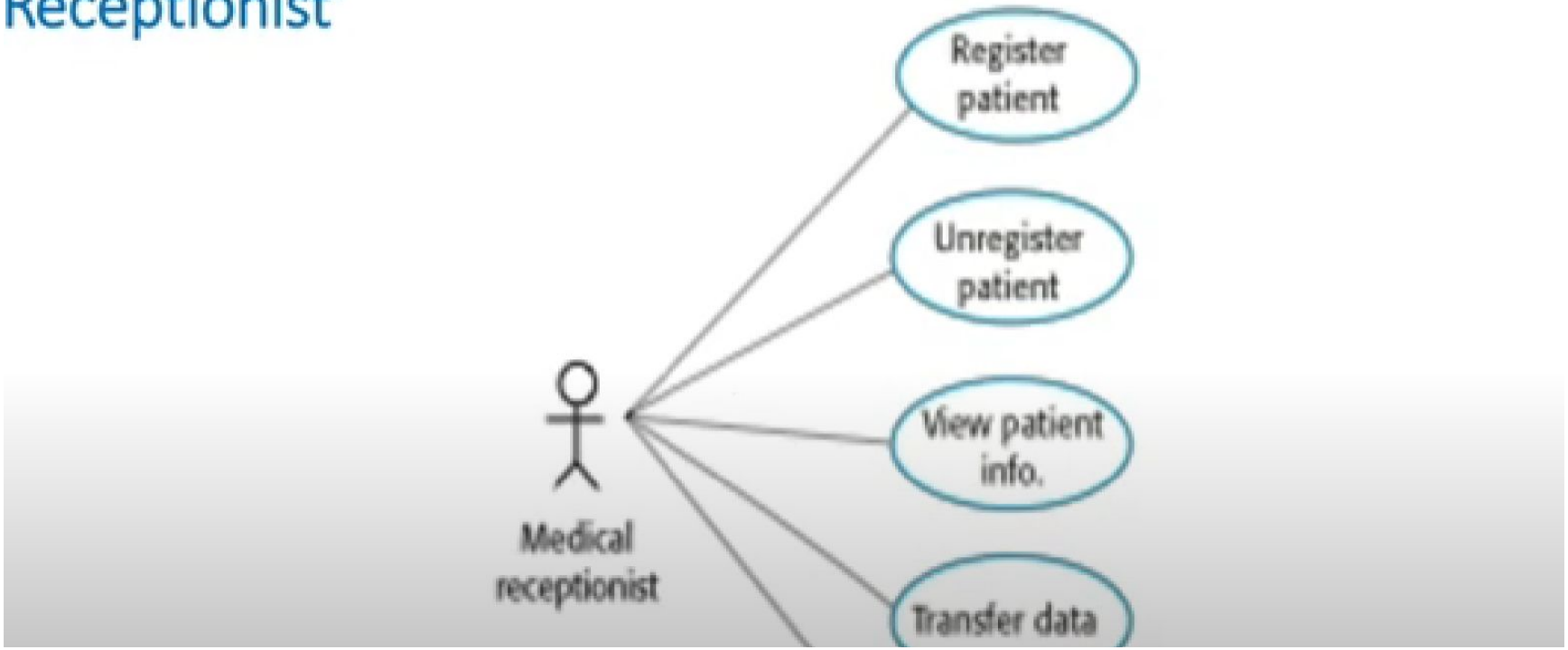
- A use case in the MHC-PMS



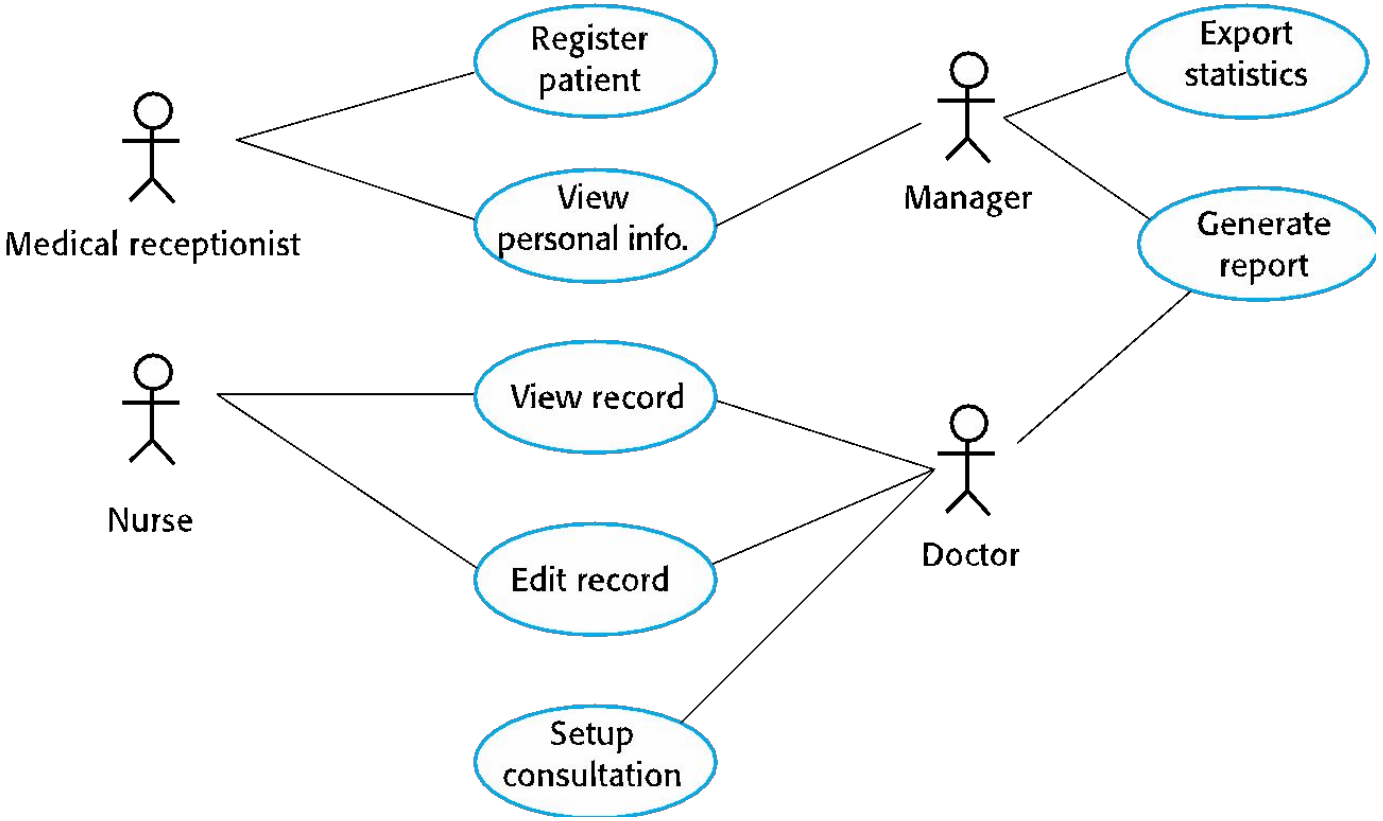
Tabular description of the 'Transfer data' use-case

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.

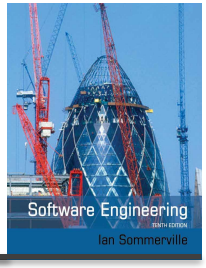
Use cases in the MHC-PMS involving the role 'Medical Receptionist'



Use cases for the Mentcare system

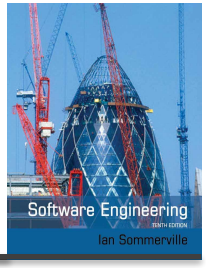


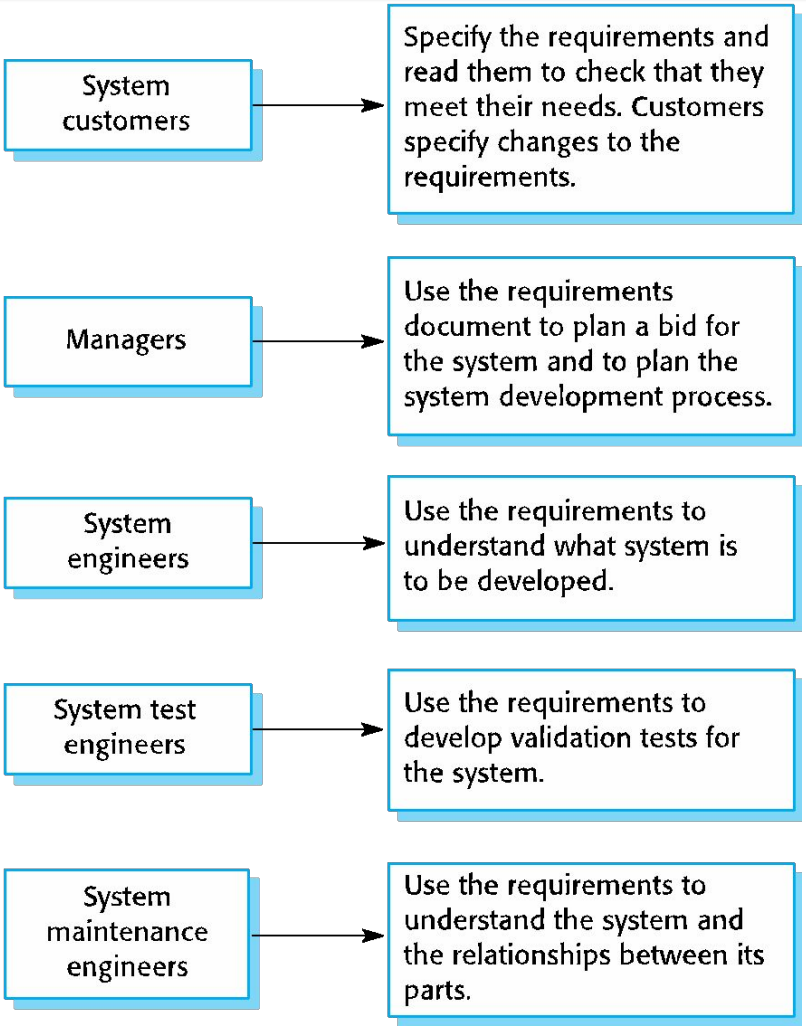
The software requirements document



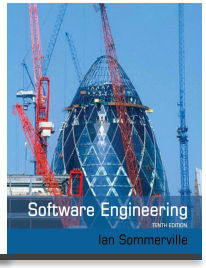
- The software requirements document is the official statement of what is required of the system developers.
- Should include both a definition of user requirements and a specification of the system requirements.
- It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it.

Users of a requirements document



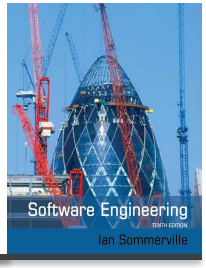


Requirements document variability



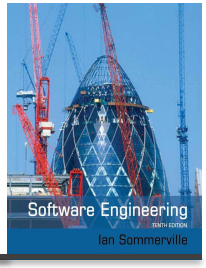
- Information in requirements document depends on type of system and the approach to development used.
- Systems developed incrementally will, typically, have less detail in the requirements document.
- Requirements documents standards have been designed e.g. IEEE standard. These are mostly applicable to the requirements for large systems engineering projects.

The structure of a requirements document

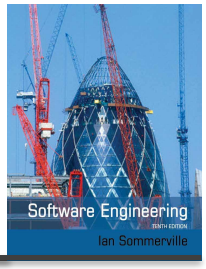


Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly 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 nonfunctional 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.

The structure of a requirements document

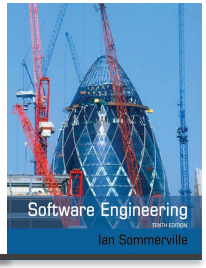


Chapter	Description
System requirements specification	This should describe the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components and 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.



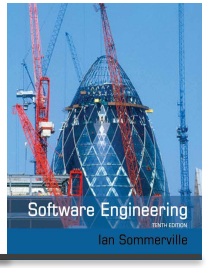
Requirements validation

Requirements validation



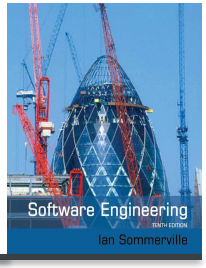
- Concerned with demonstrating that the requirements define the system that the customer really wants.
- Requirements error costs are high so validation is very important
  - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements checking



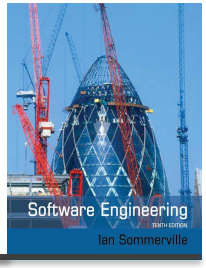
- Validity. Does the system provide the functions which best support the customer's needs?
- Consistency. Are there any requirements conflicts?
- Completeness. Are all functions required by the customer included?
- Realism. Can the requirements be implemented given available budget and technology
- Verifiability.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

Requirements validation techniques



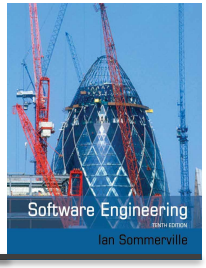
- Requirements reviews
  - Systematic manual analysis of the requirements.
- Prototyping
  - Using an executable model of the system to check requirements. Covered in Chapter 2.
- Test-case generation
  - Developing tests for requirements to check testability.

Requirements reviews

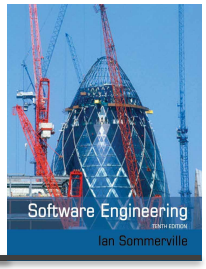


- Regular reviews should be held while the requirements definition is being formulated.
- Both client and contractor staff should be involved in reviews.
- Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.

Review checks

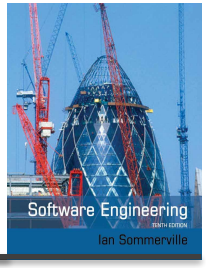


- Verifiability
  - Is the requirement realistically testable?
- Comprehensibility
  - Is the requirement properly understood?
- Traceability
  - Is the origin of the requirement clearly stated?
- Adaptability
  - Can the requirement be changed without a large impact on other requirements?



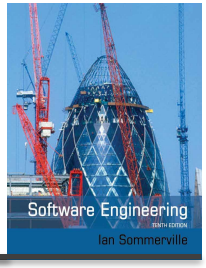
Requirements change

Changing requirements



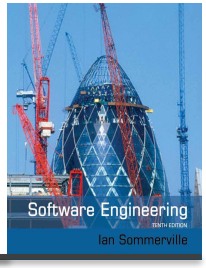
- The business and technical environment of the system always changes after installation.
  - New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.
- The people who pay for a system and the users of that system are rarely the same people.
  - System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery, new features may have to be added for user support if the system is to meet its goals.

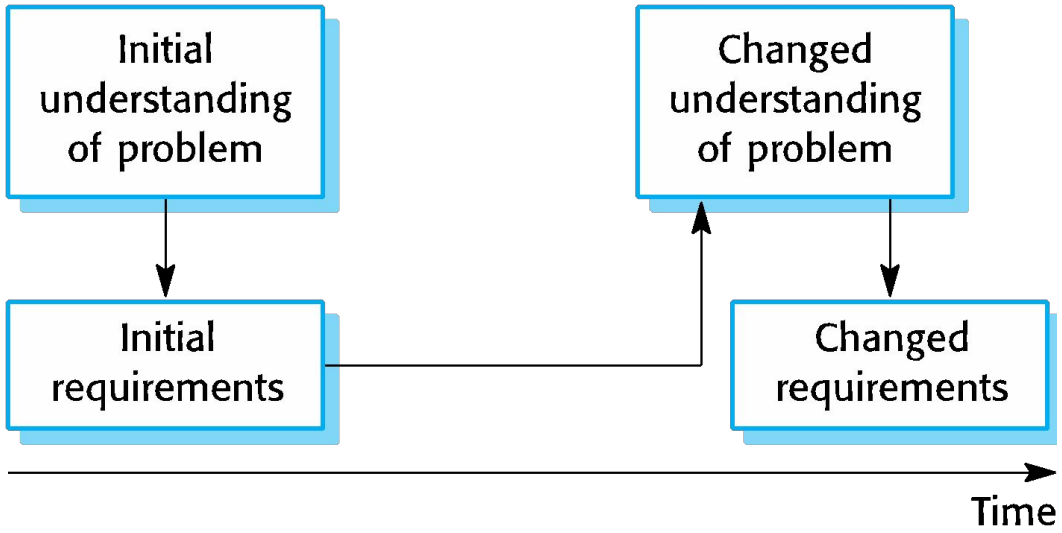
Changing requirements



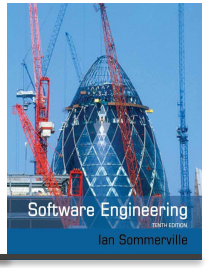
- Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.
  - The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

Requirements evolution



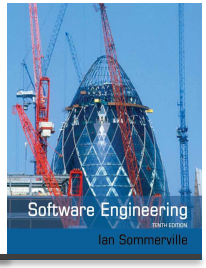


Requirements management



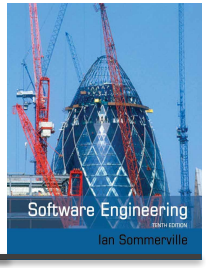
- Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- New requirements emerge as a system is being developed and after it has gone into use.
- You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You need to establish a formal process for making change proposals and linking these to system requirements.

Requirements management planning



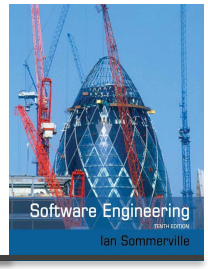
- Establishes the level of requirements management detail that is required.
- Requirements management decisions:
  - *Requirements identification* Each requirement must be uniquely identified so that it can be cross-referenced with other requirements.
  - *A change management process* This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
  - *Traceability policies* These policies define the relationships between each requirement and between the requirements and the system design that should be recorded.
  - *Tool support* Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

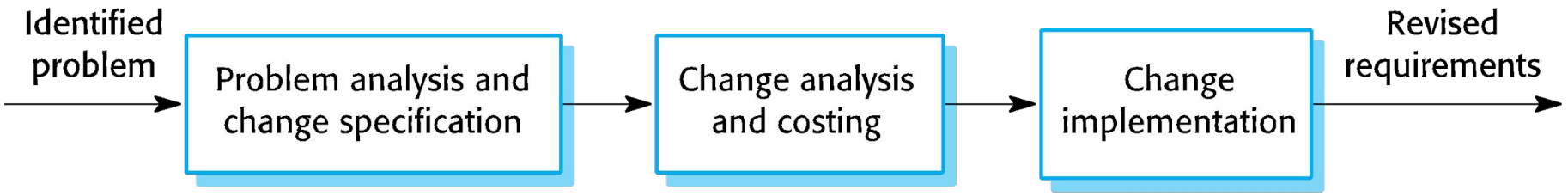
Requirements change management



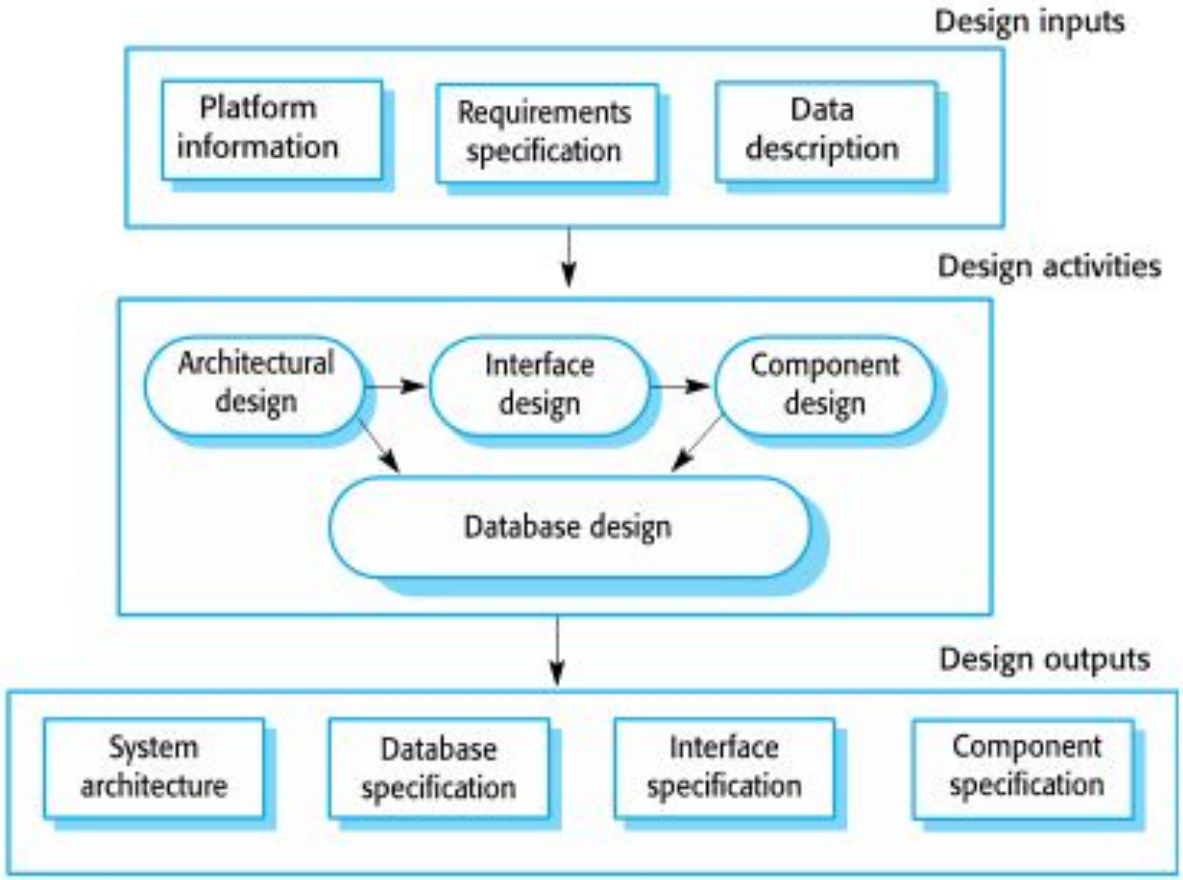
- Deciding if a requirements change should be accepted
  - *Problem analysis and change specification*
    - During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
  - *Change analysis and costing*
    - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
  - Change implementation
    - The requirements document and, where necessary, the system design and implementation, are modified. Ideally, the document should be organized so that changes can be easily implemented.

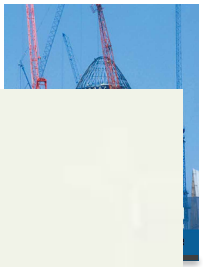
Requirements change management





A general model of the design process





Why Architecture?

- Architecture serves as a **blueprint for a system**. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.
- Architecture of a system describes the components and how they fit together.
- The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

(1) analyze the effectiveness of the design in meeting its stated requirements,

(2) consider architectural alternatives at a stage when making design changes is still relatively easy, and

(3) reduce the risks associated with the construction of the software.

Why is Architecture Important?

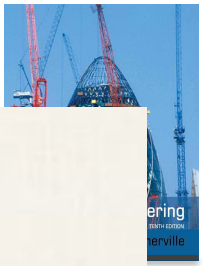
- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture "constitutes a relatively small, intellectually graspable mode of how the system is structured and how its components work together" [BAS03].

Architectural Genres

- *Genre* implies a specific category within the overall software domain.
- Within each category, you encounter a number of subcategories.
  - For example, within the genre of *buildings*, you would encounter the following general styles: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on.
  - Within each general style, more specific styles might apply. Each style would have a structure that can be described using a set of predictable patterns.

Genre Examples for Software Systems

- Artificial Intelligence
- Devices
- Sports
- Financial
- Games
- Medical
- Scientific
- Transportation
- Government
- Etc.



Architectural Styles

- There's a pattern or type of architecture at the back of each artist.
  - Differentiate a house from other styles
- Software also exhibits some styles
- Each style describes a system category that encompasses:

(1) **set of components** (e.g., a database, computational modules) that perform a function required by a system,

(2) **set of connectors** that enable "communication, coordination and cooperation" among components,

(3) **constraints** that define how components can be integrated to form the system, and

(4) **semantic models** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

Taxonomy of Styles in Software

- Data-centered architectures
  - There is a central data server which is accessed by clients
- Data flow architectures
  - Data travels through a series of components
- Call and return architectures
  - Classical
- Object-oriented architectures
  - Modern style. Components pass messages
- Layered architectures
  - High-level to machine level

1. Data-Centered Architecture

- The data stored in the file or database is occupying at the center of the architecture.
- Store data is access continuously by the other components like an update, delete, add, modify from the data store.
- Data-centered architecture helps integrity.
- Pass data between clients using the blackboard mechanism.
- The processes are independently executed by the client components.

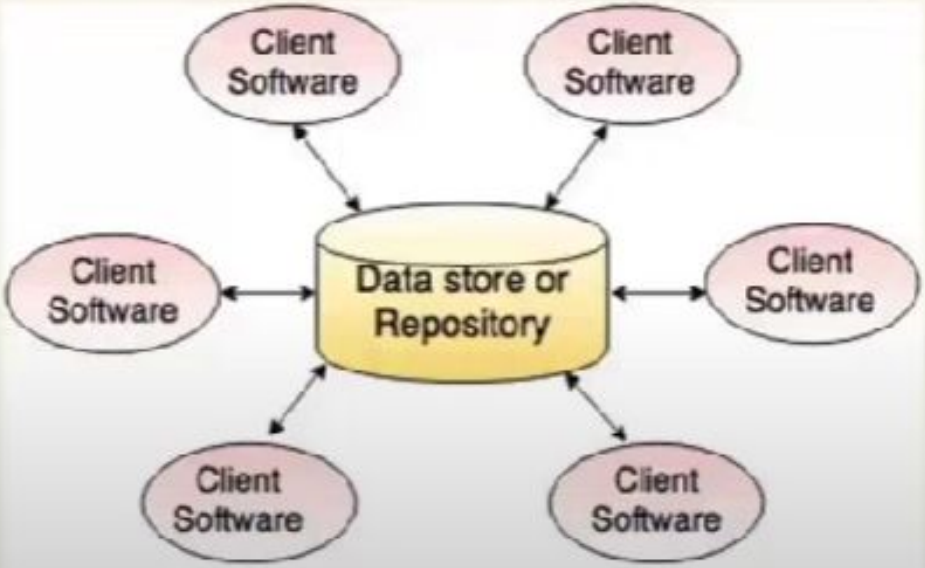
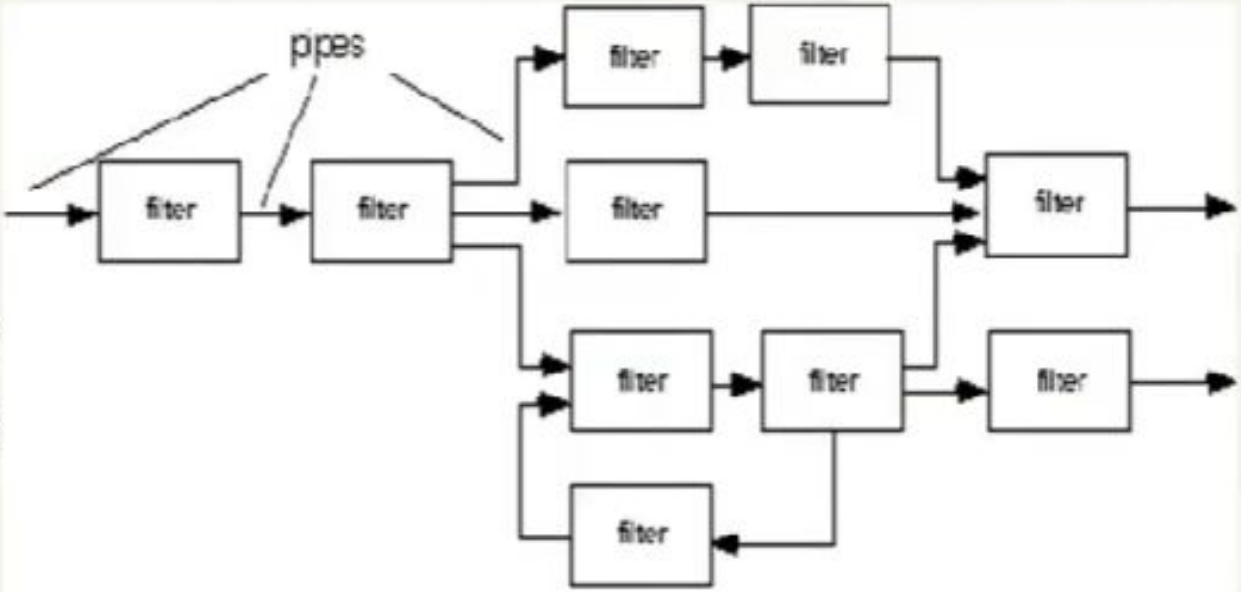


Fig.- Data centered architecture

- This architecture is applied when the input data is converted into a series of manipulative components into output data.
- A pipe and filter pattern is a set of components called as filters.
- Filters are connected through pipes and transfer data from one component to the next component.
- The flow of data degenerates into a single line of transform then it is known as batch sequential.

2. Data Flow Architecture



(a) pipes and filters

3. Call and Return Architecture

This architecture style allows to achieve a program structure which is easy to modify.

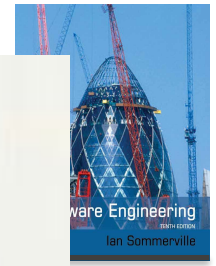
Following are the sub styles exist in this category:

1. Main program or subprogram architecture

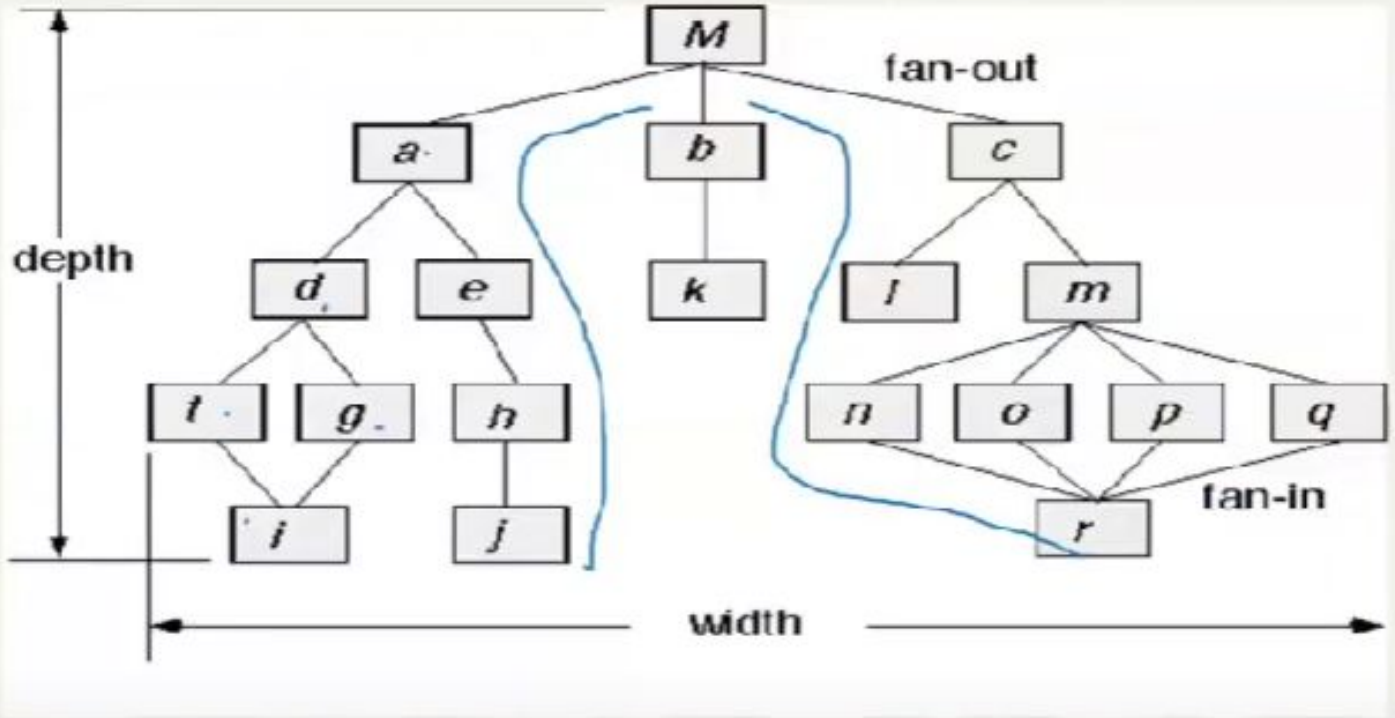
- The program is divided into smaller pieces hierarchically.
- The main program invokes many of program components in the hierarchy that program components are divided into subprogram.

2. Remote procedure call architecture

- The main program or subprogram components are distributed in network of multiple computers.
- The main aim is to increase the performance.



3. Call and Return Architecture...



Structural metrics which measure inter-module complexities.

Fan-in: Is the number of modules that call a given module.

Fan-out: Is the numbers of modules that called by a given module

4. Layered Architecture

- The different layers are defined in the architecture. It consists of outer and inner layer.
- The components of outer layer manage the user interface operations.
- Components execute the operating system interfacing at the inner layer.
- The inner layers are application layer, utility layer and the core layer.
- In many cases, It is possible that more than one pattern is suitable and the alternate architectural style can be designed and evaluated.

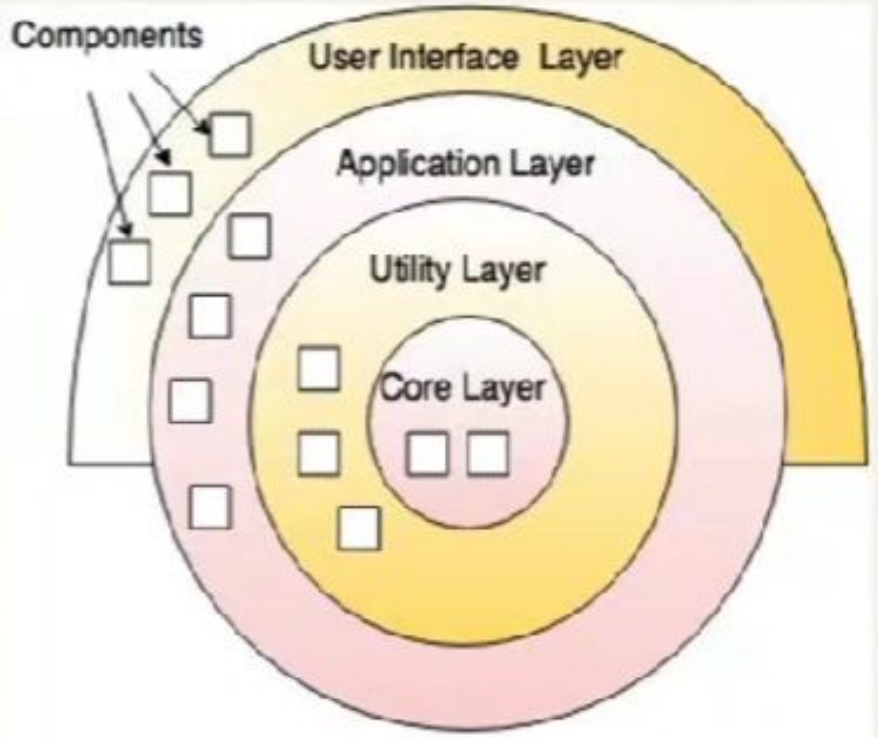


Fig.- Layered Architecture

Architectural Patterns

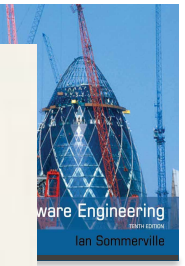
- Design solutions to recurring problems

Examples:

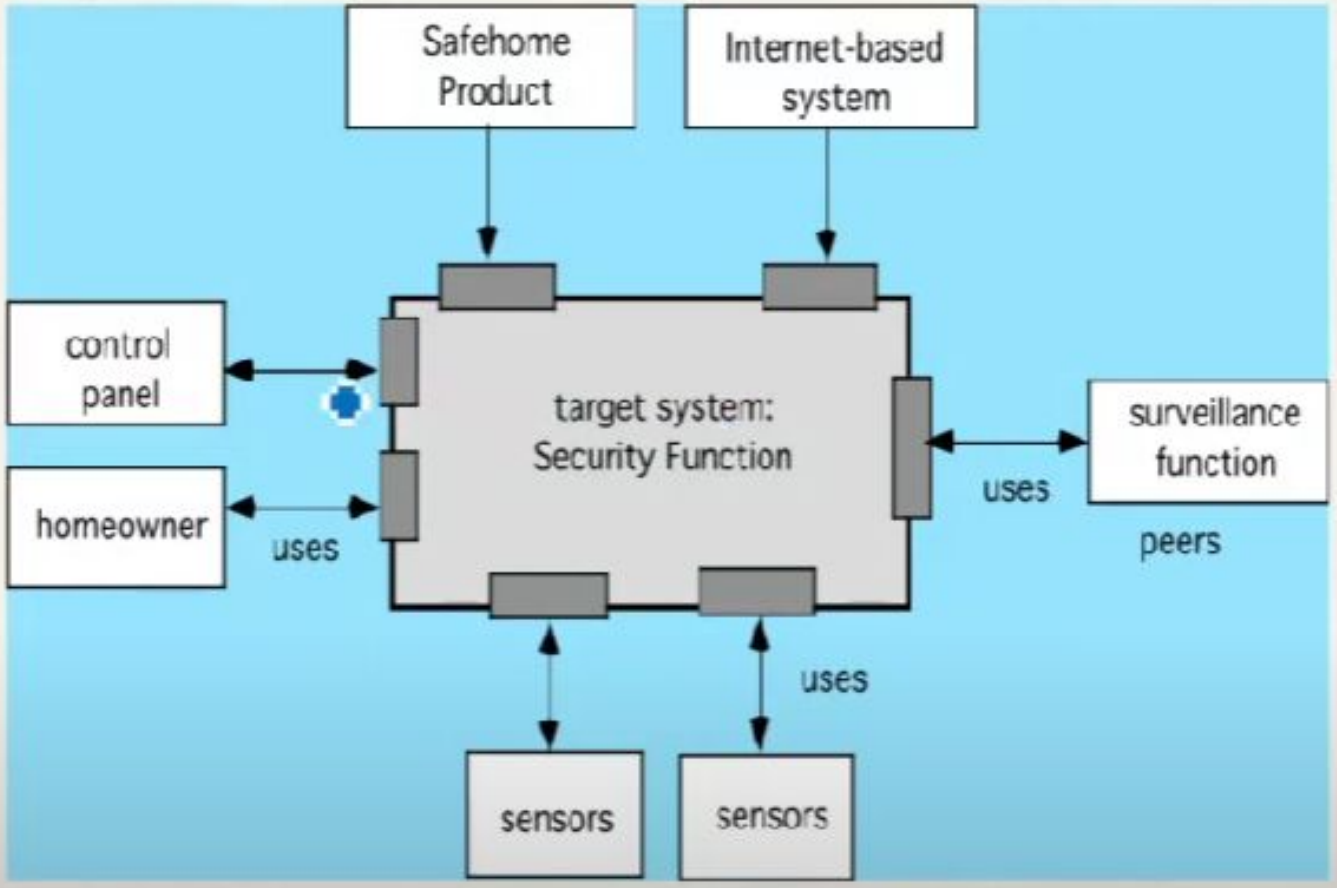
- Concurrency—applications must handle multiple tasks in a manner that simulates parallelism
  - *operating system process management* pattern
  - *task scheduler* pattern
- Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:
  - a *database management system* pattern that applies the storage and retrieval capability of a DBMS to the application architecture
  - an *application level persistence* pattern that builds persistence features into the application architecture
- Distribution— the manner in which systems or components within systems communicate with one another in a distributed environment
  - A *broker* acts as a 'middle-man' between the client component and a server component.

Architectural Design

- The software must be placed into context
  - the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction
- A set of architectural archetypes should be identified
  - An *archetype* is an abstraction (similar to a class) that represents one element of system behavior
- The designer specifies the structure of the system by defining and refining software components that implement each archetype



Architectural Context



Archetypes

Abstract Building Blocks

- Node
  - Input, Output
- Controller
  - Arms-disarms a node
- Detector
  - Sensing
- Indicator
  - Alarms

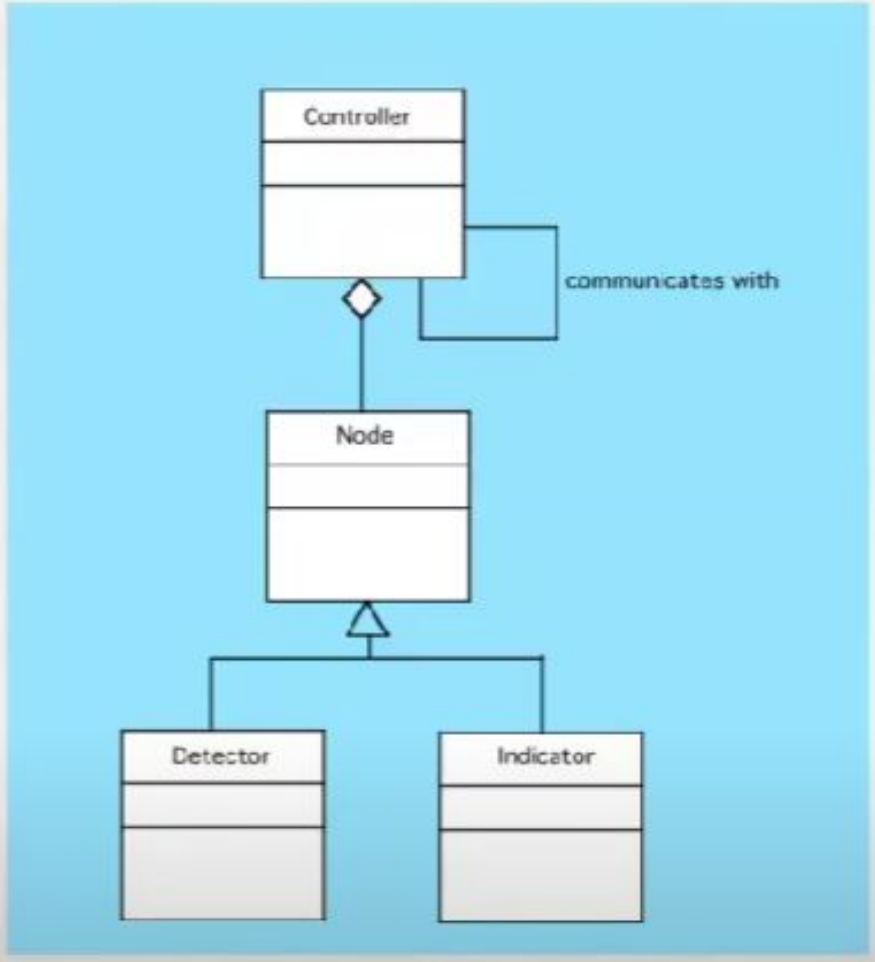
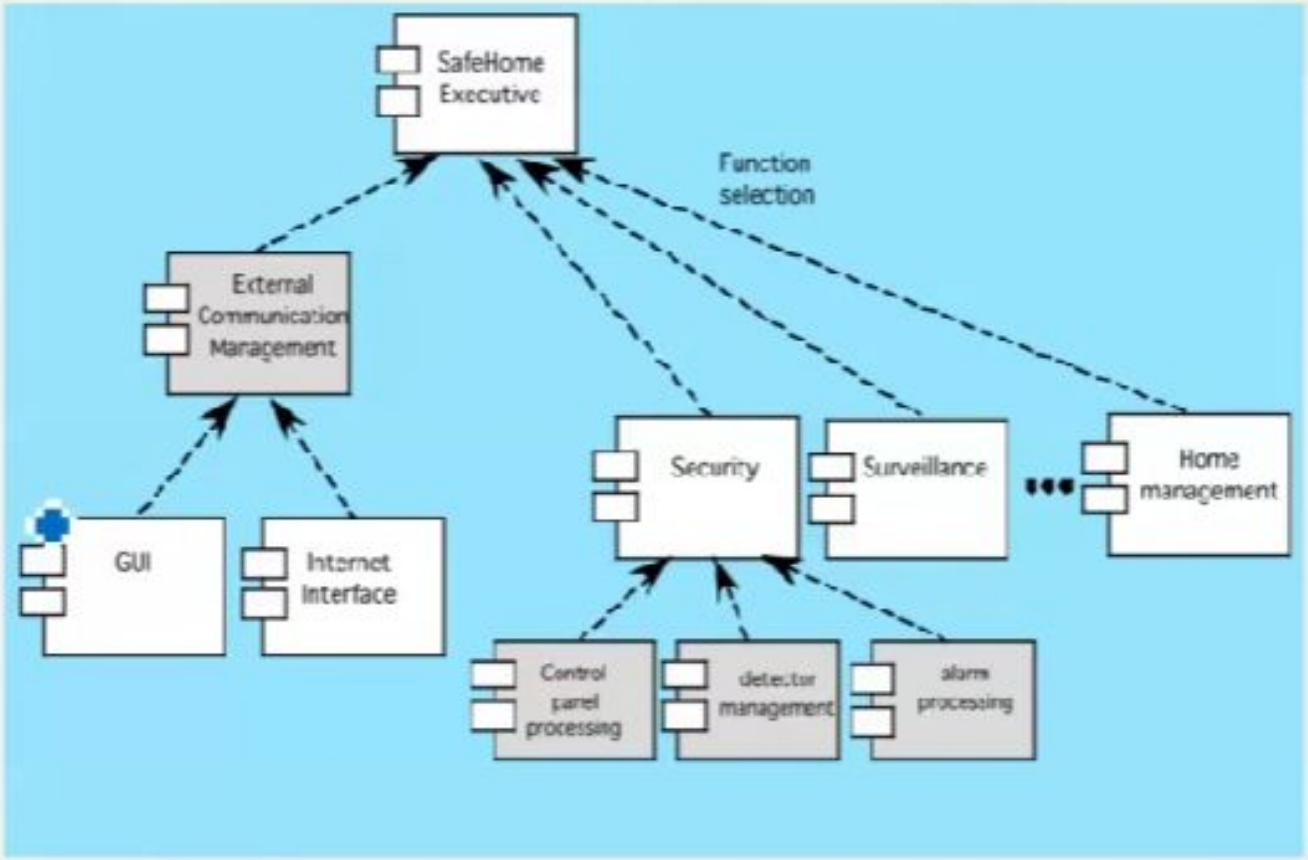
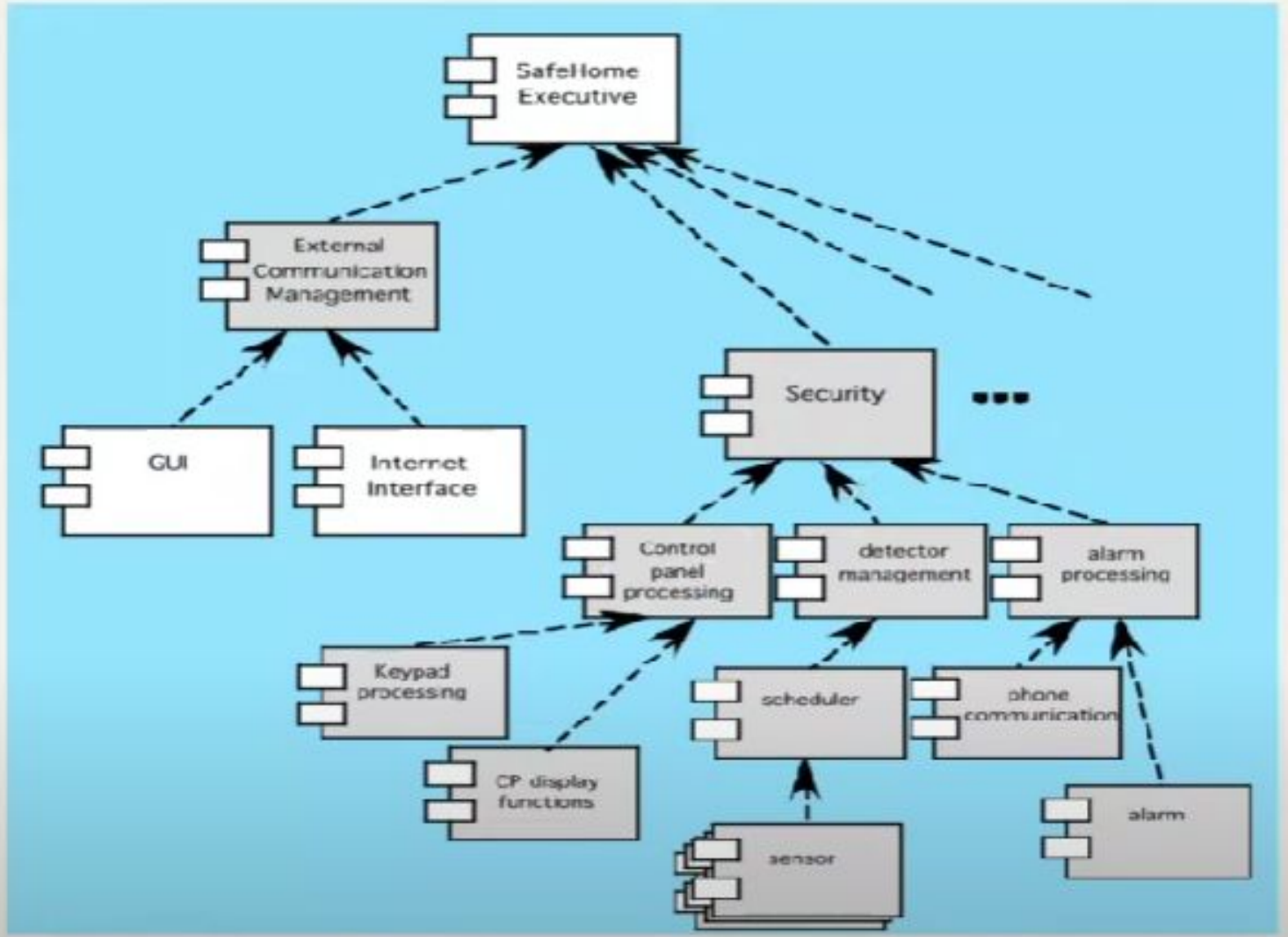


Figure 10.7 UML relationships for SafeHome security function archetypes (adapted from [BOS00])

Component Structure



Refined Component Structure



What is a Component?

- *OMG Unified Modeling Language Specification* [OMG01] defines a component as
  - "... a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."
- *OO view:* a component contains a set of collaborating classes
- *Conventional view:* a component contains processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.

OO Component

- During requirement engineering, an analysis class called PrintJob was derived.
- During architectural design, PrintJob is defined as a component within the system architecture.
- Component-level design elaborate PrintJob to provide sufficient information to guide implementation.

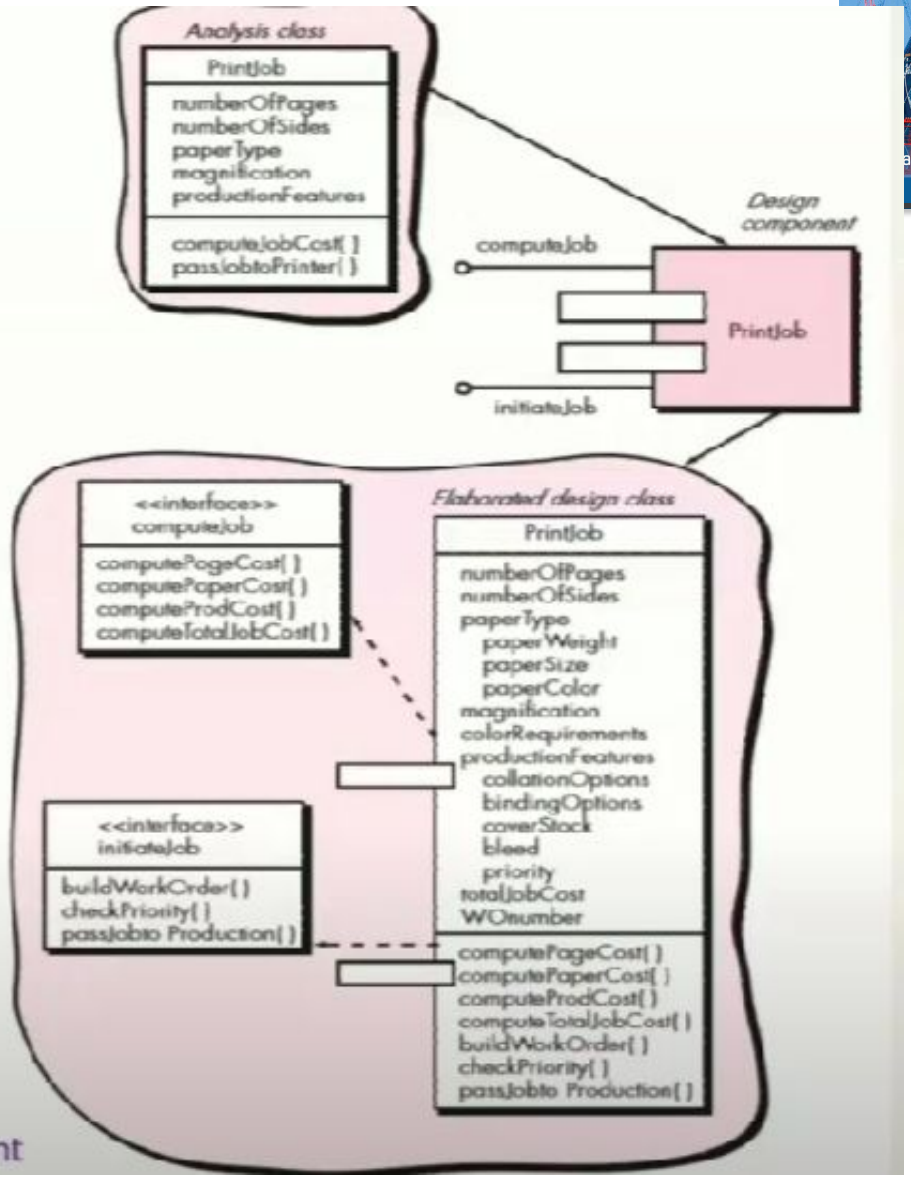
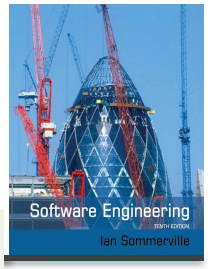
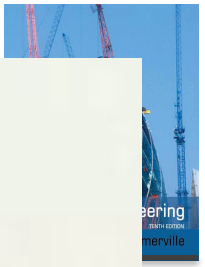


Fig: Elaboration of a design component



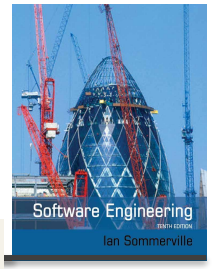
Basic Design Principles

- The Open-Closed Principle (OCP). *"A module [component] should be open for extension but closed for modification.*
- The Liskov Substitution Principle (LSP). *"Subclasses should be substitutable for their base classes.*
- Dependency Inversion Principle (DIP). *"Depend on abstractions. Do not depend on concretions."*
- The Interface Segregation Principle (ISP). *"Many client-specific interfaces are better than one general purpose interface.*
- The Release Reuse Equivalency Principle (REP). *"The granule of reuse is the granule of release."*
- The Common Closure Principle (CCP). *"Classes that change together belong together."*
- The Common Reuse Principle (CRP). *"Classes that aren't reused together should not be grouped together."*



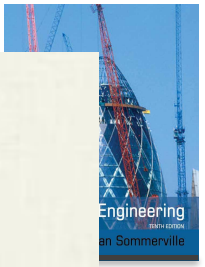
Design Guidelines

- Components
  - Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
- Interfaces
  - Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OPC)
- Dependencies and Inheritance
  - it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).



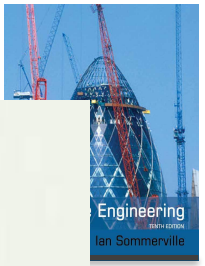
Cohesion

- **Conventional view:**
  - the "single-mindedness" of a module
- OO view:
  - *cohesion* implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- Levels of cohesion
  - Functional
  - Layer
  - Communicational
  - Sequential
  - Procedural
  - Temporal
  - utility



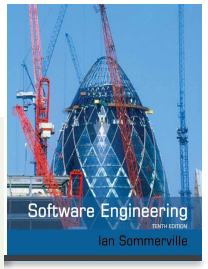
Coupling

- Conventional view:
  - The degree to which a component is connected to other components and to the external world
- OO view:
  - a qualitative measure of the degree to which classes are connected to one another
- Level of coupling
  - Content
  - Common
  - Control
  - Stamp
  - Data
  - Routine call
  - Type use
  - Inclusion or import
  - External



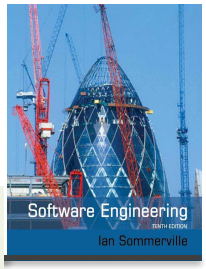
Component Level Design-I

- Step 1. Identify all design classes that correspond to the problem domain.
- Step 2. Identify all design classes that correspond to the infrastructure domain.
- Step 3. Elaborate all design classes that are not acquired as reusable components.
- Step 3a. Specify message details when classes or component collaborate.
- Step 3b. Identify appropriate interfaces for each component.



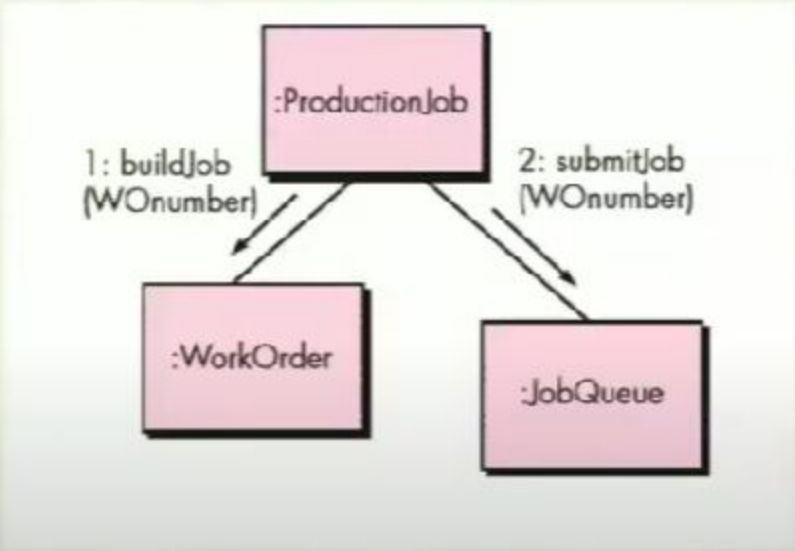
Component-Level Design-II

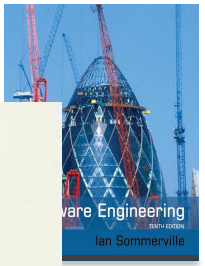
- Step 3c. Elaborate attributes and define data types and data structures required to implement them.
- Step 3d. Describe processing flow within each operation in detail.
- Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.
- Step 5. Develop and elaborate behavioral representations for a class or component.
- Step 6. Elaborate deployment diagrams to provide additional implementation detail.
- Step 7. Factor every component-level design representation and always consider alternatives.



Collaboration Diagram

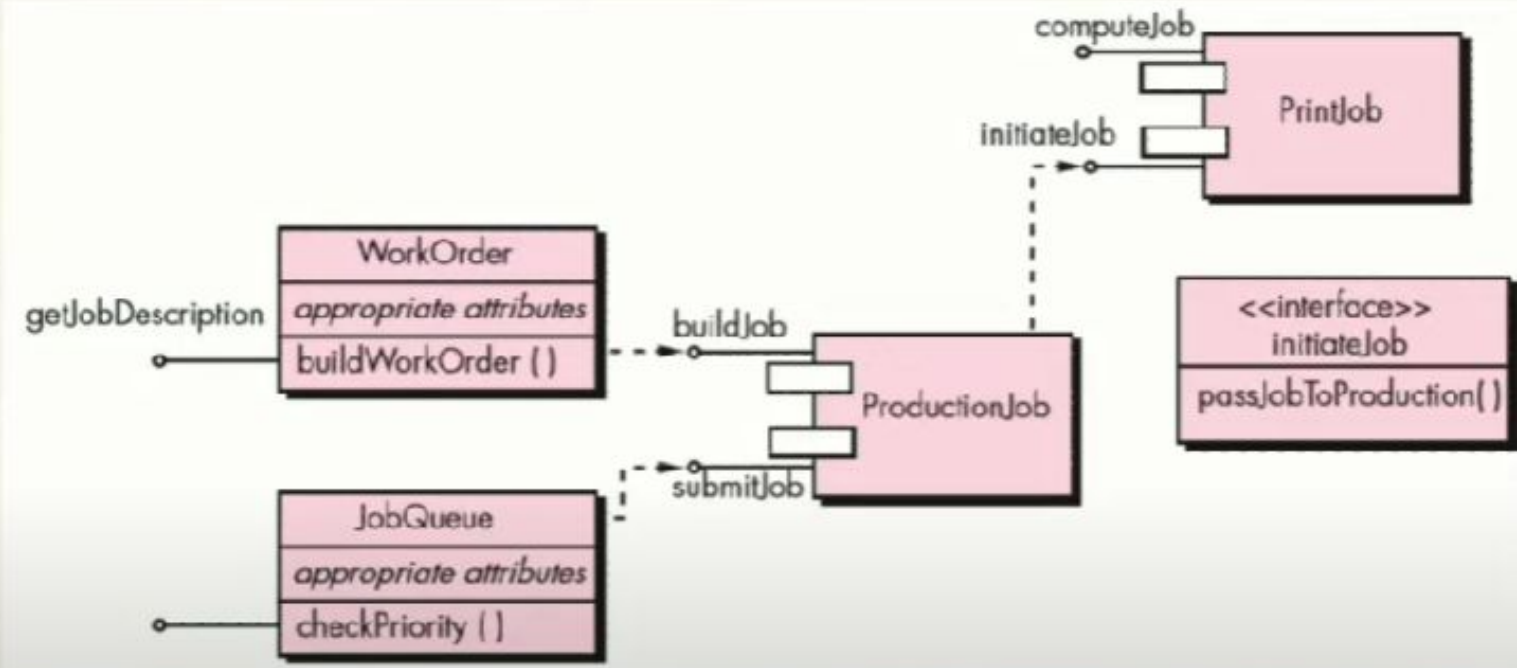
Collaboration to prepare a print job for submission to the production stream. Messages are passed between the objects.



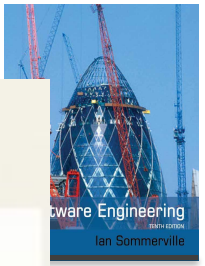


Refactoring

Refactoring interfaces and class definitions for PrintJob
Three different sub-functions are refactored into three classes, each of which focuses on one function.



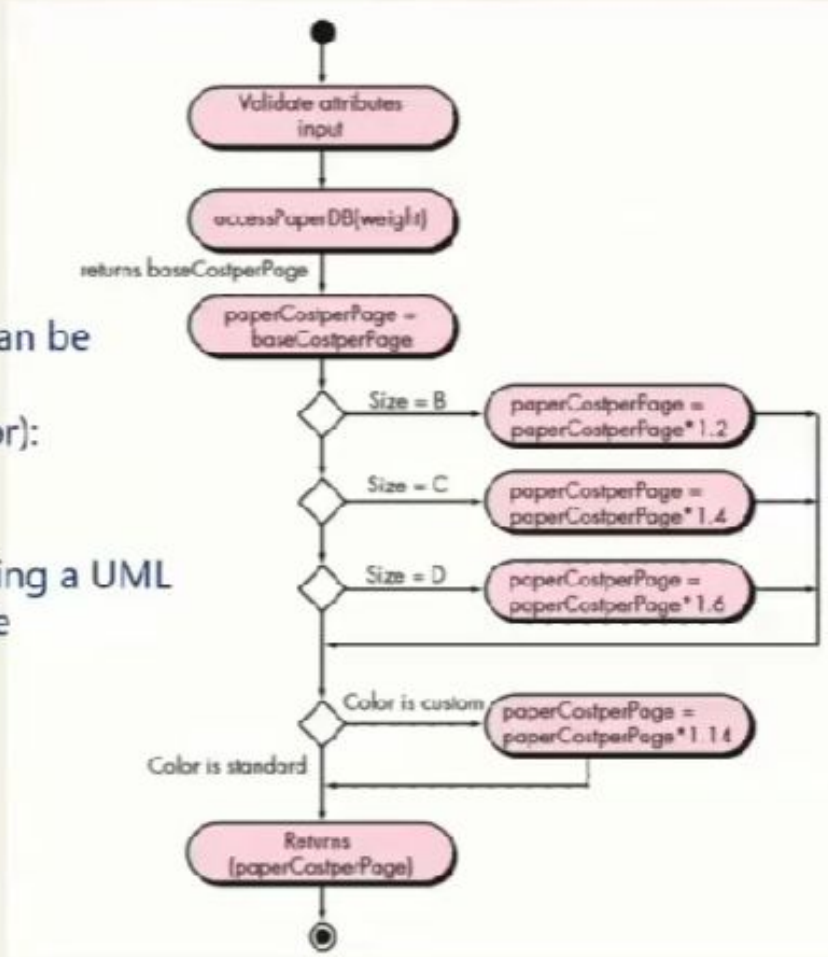
Refactoring interfaces and class definitions for PrintJob



Activity Diagram

The computePaperCost() operation can be expanded into the following:
computePaperCost (weight, size, color): numeric

The algorithm can be represented using a UML activity diagram or using pseudocode



Software Engineering: A Practitioner's Approach, 7/e

Statechart

The transition from one state to another occurs as a consequence of an event

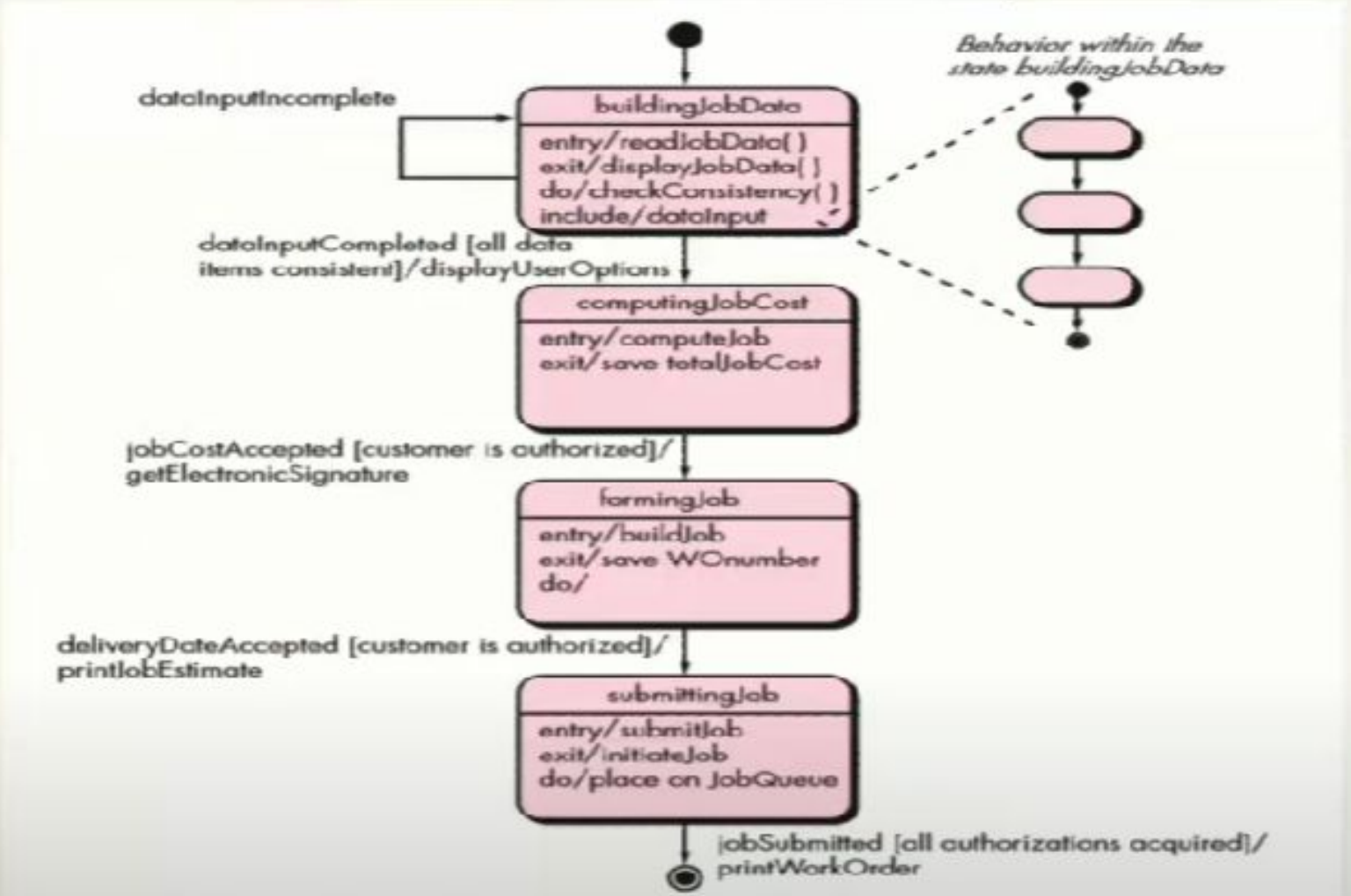
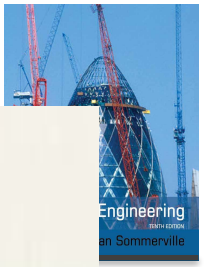
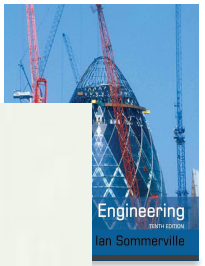


Fig: Statechart fragment for PrintJob class



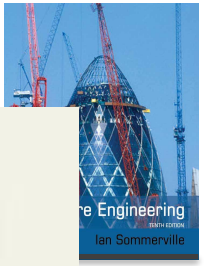
Component Design for WebApps

- WebApp component is
  - (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end-user, or
  - (2) a cohesive package of content and functionality that provides end-user with some required capability.
- Therefore, component-level design for WebApps often incorporates elements of content design and functional design.



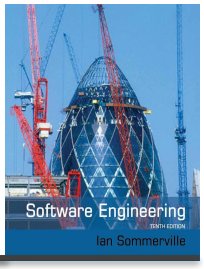
Content Design for WebApps

- focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end-user
- consider a Web-based video surveillance capability within **SafeHomeAssured.com**
  - potential content components can be defined for the video surveillance capability:
    - (1) the content objects that represent the space layout (the floor plan) with additional icons representing the location of sensors and video cameras;
    - (2) the collection of thumbnail video captures (each an separate data object), and
    - (3) the streaming video window for a specific camera.
  - Each of these components can be separately named and manipulated as a package.



Functional Design for WebApps

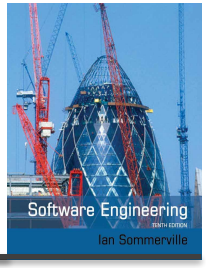
- Modern Web applications deliver increasingly sophisticated processing functions that:
  - (1) perform localized processing to generate content and navigation capability in a dynamic fashion;
  - (2) provide computation or data processing capability that is appropriate for the WebApp's business domain;
  - (3) provide sophisticated database query and access, or
  - (4) establish data interfaces with external corporate systems.
- To achieve these (and many other) capabilities, you will design and construct WebApp functional components that are identical in form to software components for conventional software.



Designing Conventional Components

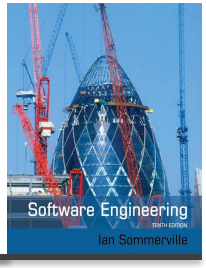
- The design of processing logic is governed by the basic principles of algorithm design and structured programming
- The design of data structures is defined by the data model developed for the system
- The design of interfaces is governed by the collaborations that a component must effect

Key points



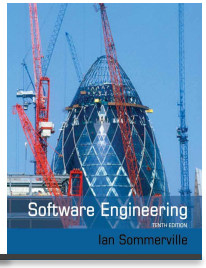
- Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.
- Non-functional requirements often constrain the system being developed and the development process being used.
- They often relate to the emergent properties of the system and therefore apply to the system as a whole.

Key points



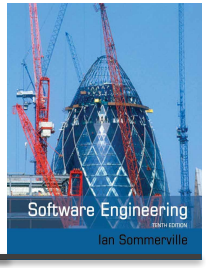
- The requirements engineering process is an iterative process that includes requirements elicitation, specification and validation.
- Requirements elicitation is an iterative process that can be represented as a spiral of activities – requirements discovery, requirements classification and organization, requirements negotiation and requirements documentation.
- You can use a range of techniques for requirements elicitation including interviews and ethnography. User stories and scenarios may be used to facilitate discussions.

Key points

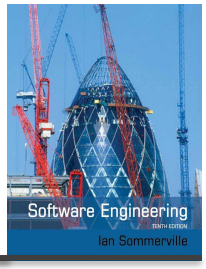


- Requirements specification is the process of formally documenting the user and system requirements and creating a software requirements document.
- The software requirements document is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.

Key points

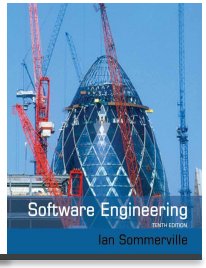


- Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism and verifiability.
- Business, organizational and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.



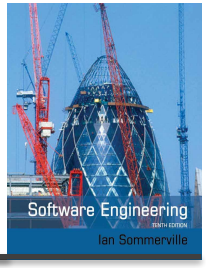
Chapter 5 – System Modeling

Topics covered



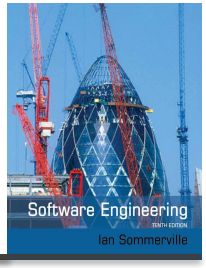
- Context models
- Interaction models
- Structural models
- Behavioral models
- Model-driven engineering

System modeling



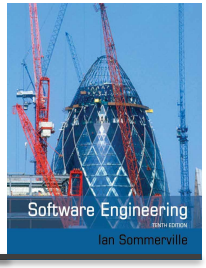
- System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.
- System modeling has now come to mean representing a system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).
- System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.

Existing and planned system models



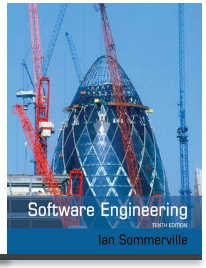
- Models of the existing system are used during requirements engineering. They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.
- Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation.
- In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.

System perspectives



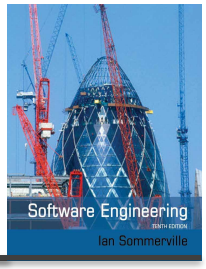
- An external perspective, where you model the context or environment of the system.
- An interaction perspective, where you model the interactions between a system and its environment, or between the components of a system.
- A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.
- A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

UML diagram types

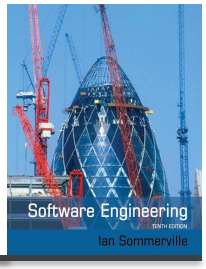


- Activity diagrams, which show the activities involved in a process or in data processing .
- Use case diagrams, which show the interactions between a system and its environment.
- Sequence diagrams, which show interactions between actors and the system and between system components.
- Class diagrams, which show the object classes in the system and the associations between these classes.
- State diagrams, which show how the system reacts to internal and external events.

Use of graphical models

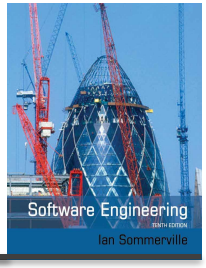


- As a means of facilitating discussion about an existing or proposed system
  - Incomplete and incorrect models are OK as their role is to support discussion.
- As a way of documenting an existing system
  - Models should be an accurate representation of the system but need not be complete.
- As a detailed system description that can be used to generate a system implementation
  - Models have to be both correct and complete.



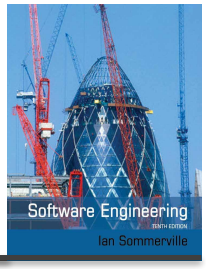
Context models

Context models



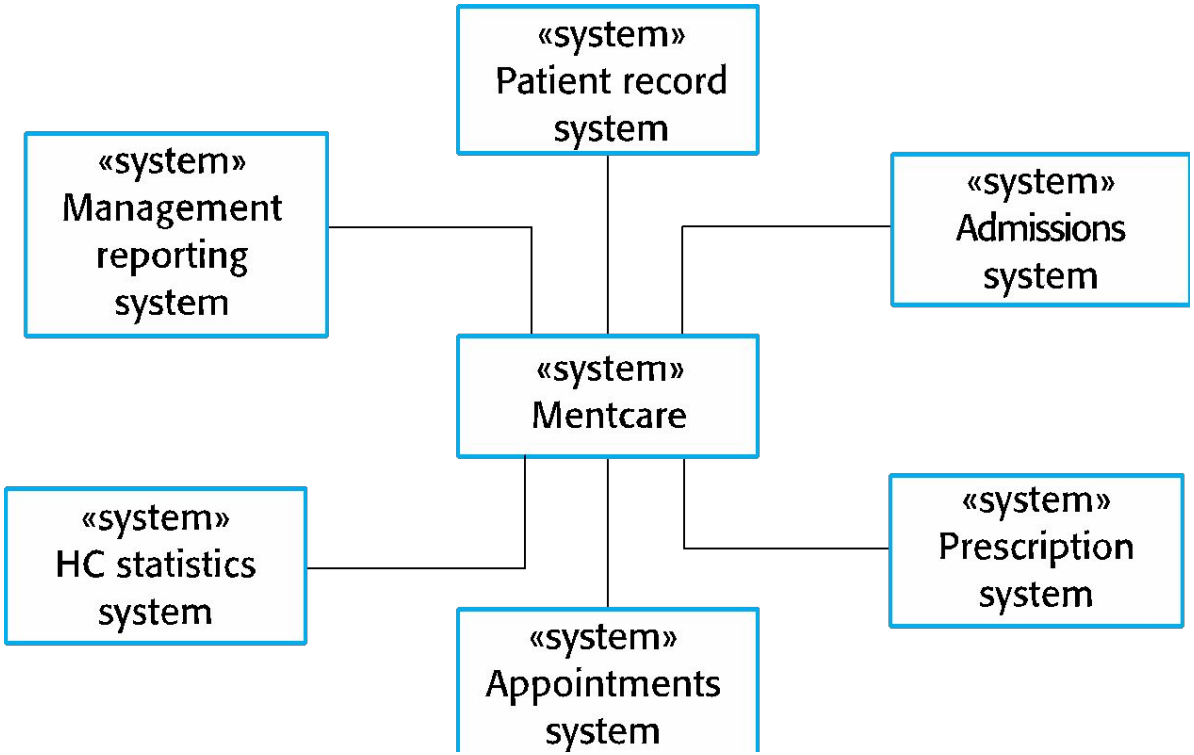
- Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
- Social and organisational concerns may affect the decision on where to position system boundaries.
- Architectural models show the system and its relationship with other systems.

System boundaries

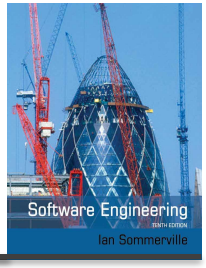


- System boundaries are established to define what is inside and what is outside the system.
  - They show other systems that are used or depend on the system being developed.
- The position of the system boundary has a profound effect on the system requirements.
- Defining a system boundary is a political judgment
  - There may be pressures to develop system boundaries that increase / decrease the influence or workload of different parts of an organization.

The context of the Mentcare system

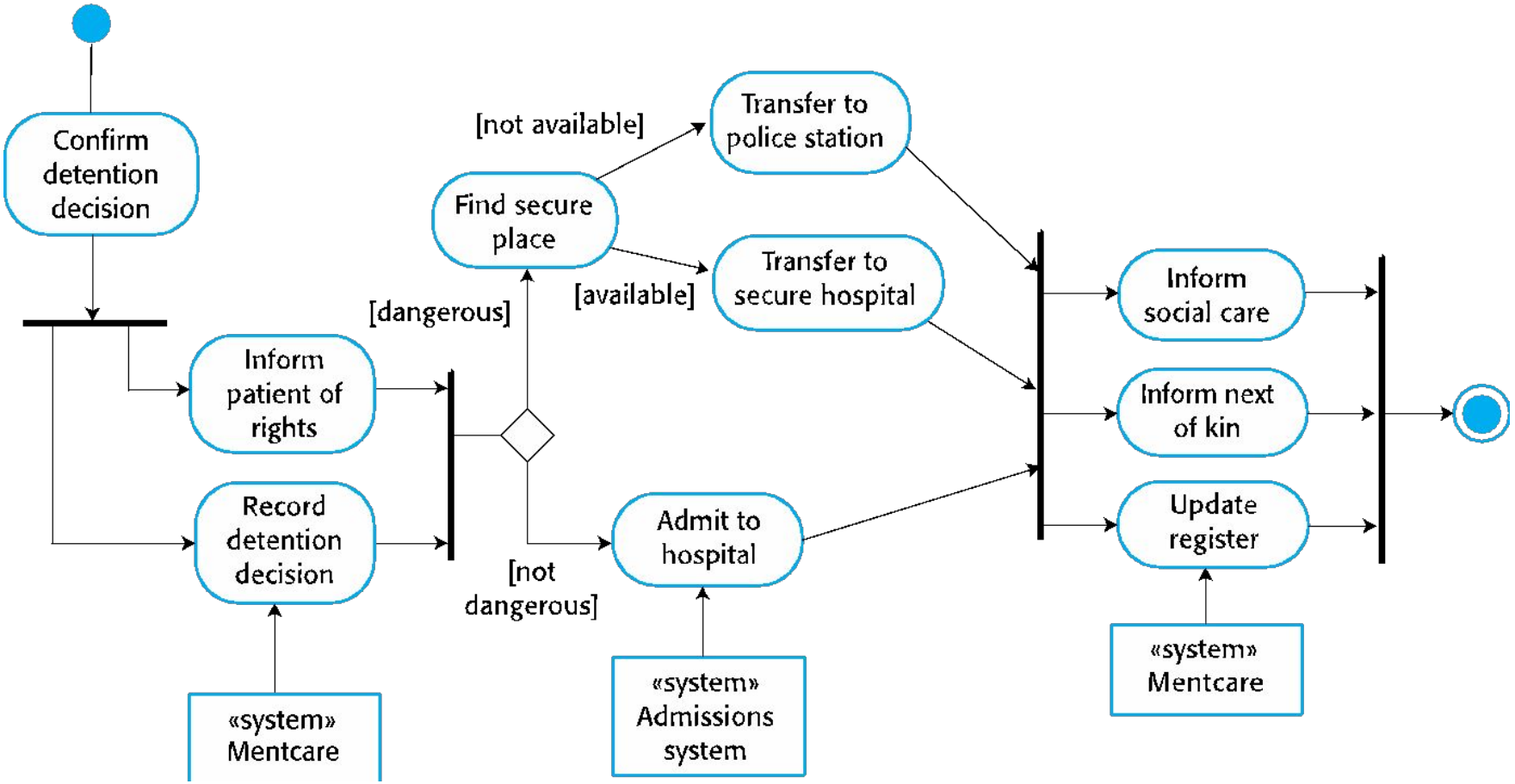


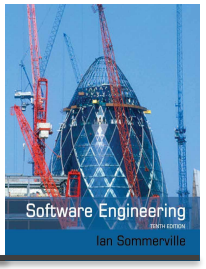
Process perspective



- Context models simply show the other systems in the environment, not how the system being developed is used in that environment.
- Process models reveal how the system being developed is used in broader business processes.
- UML activity diagrams may be used to define business process models.

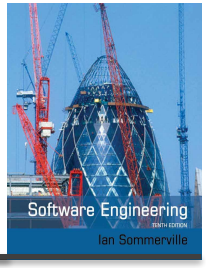
Process model of involuntary detention





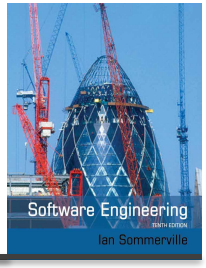
Interaction models

Interaction models



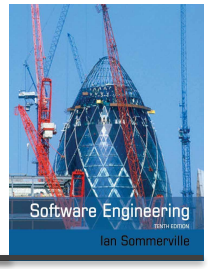
- Modeling user interaction is important as it helps to identify user requirements.
- Modeling system-to-system interaction highlights the communication problems that may arise.
- Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.
- Use case diagrams and sequence diagrams may be used for interaction modeling.

Use case modeling



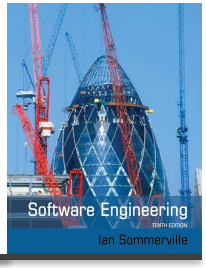
- Use cases were developed originally to support requirements elicitation and now incorporated into the UML.
- Each use case represents a discrete task that involves external interaction with a system.
- Actors in a use case may be people or other systems.
- Represented diagramatically to provide an overview of the use case and in a more detailed textual form.

Transfer-data use case



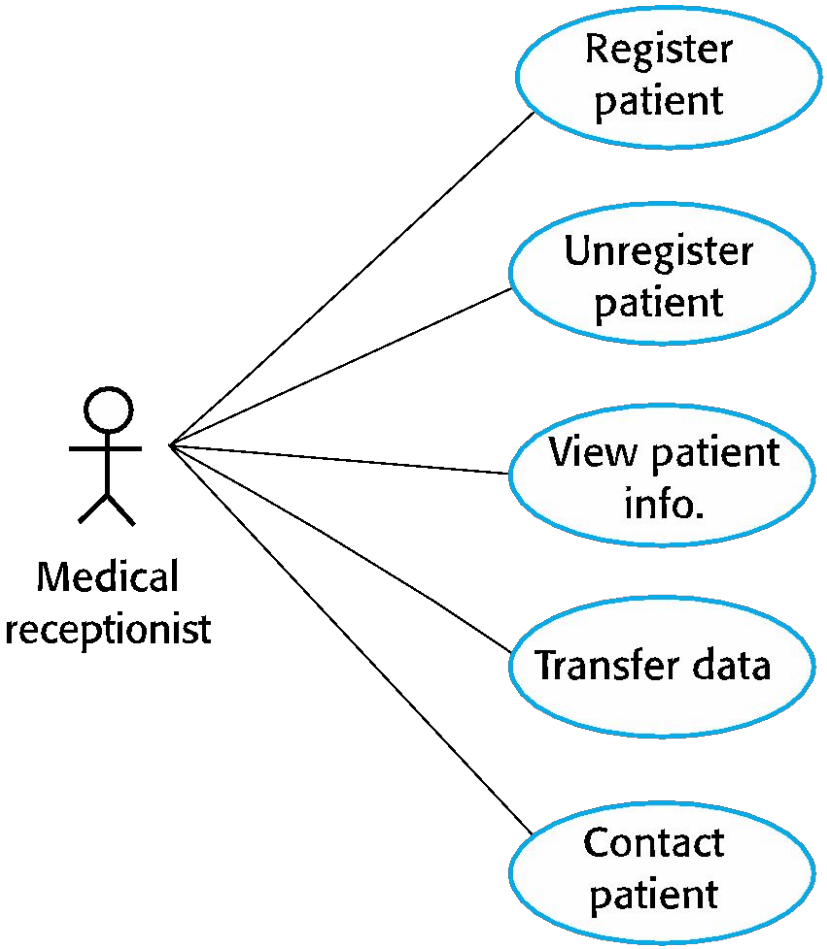
- A use case in the Mentcare system

Tabular description of the ‘Transfer data’ use-case

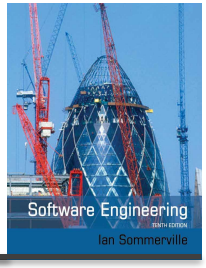


MHC-PMS: Transfer data	
Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the Mentcase system to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (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.

Use cases in the Mentcare system involving the role ‘Medical Receptionist’

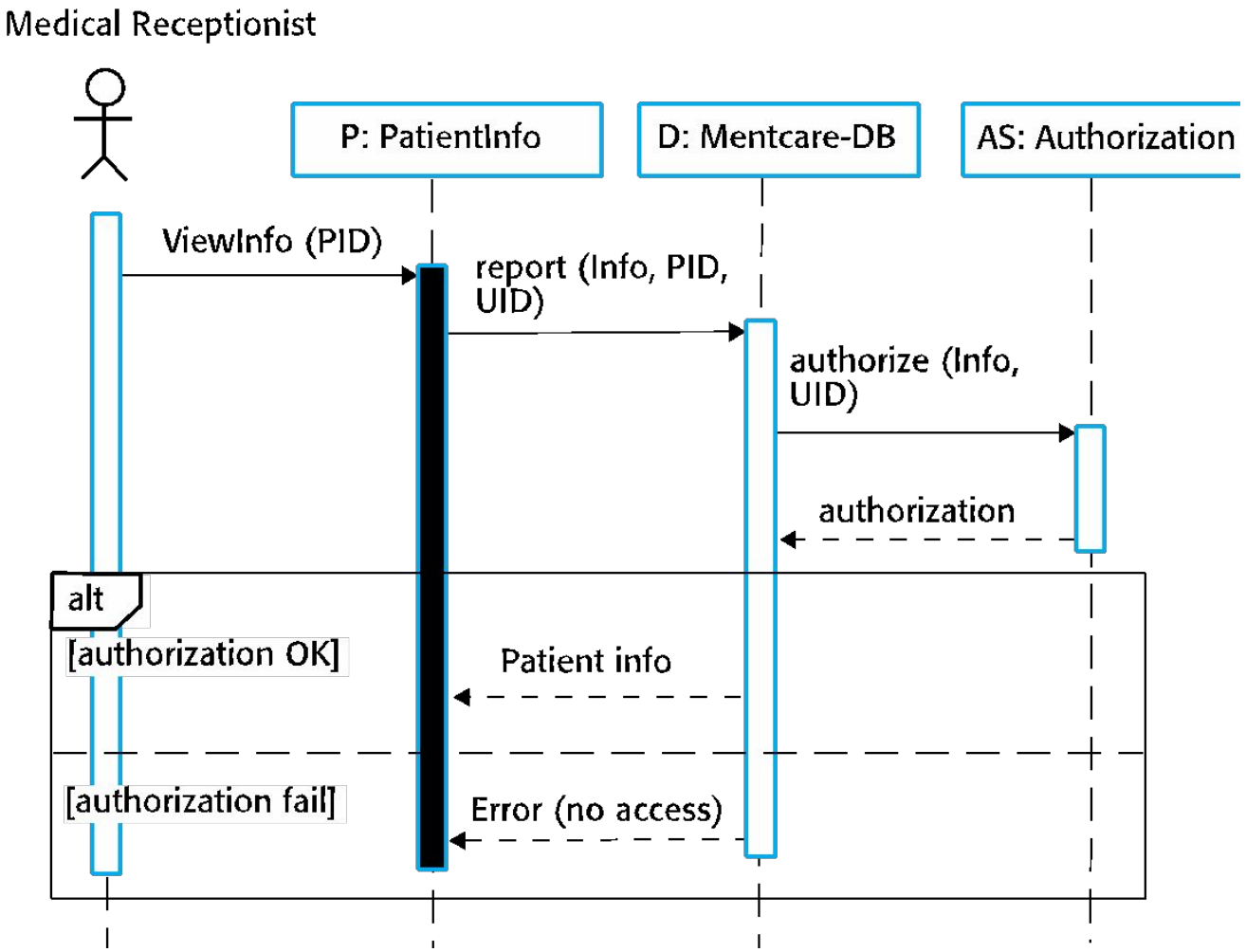


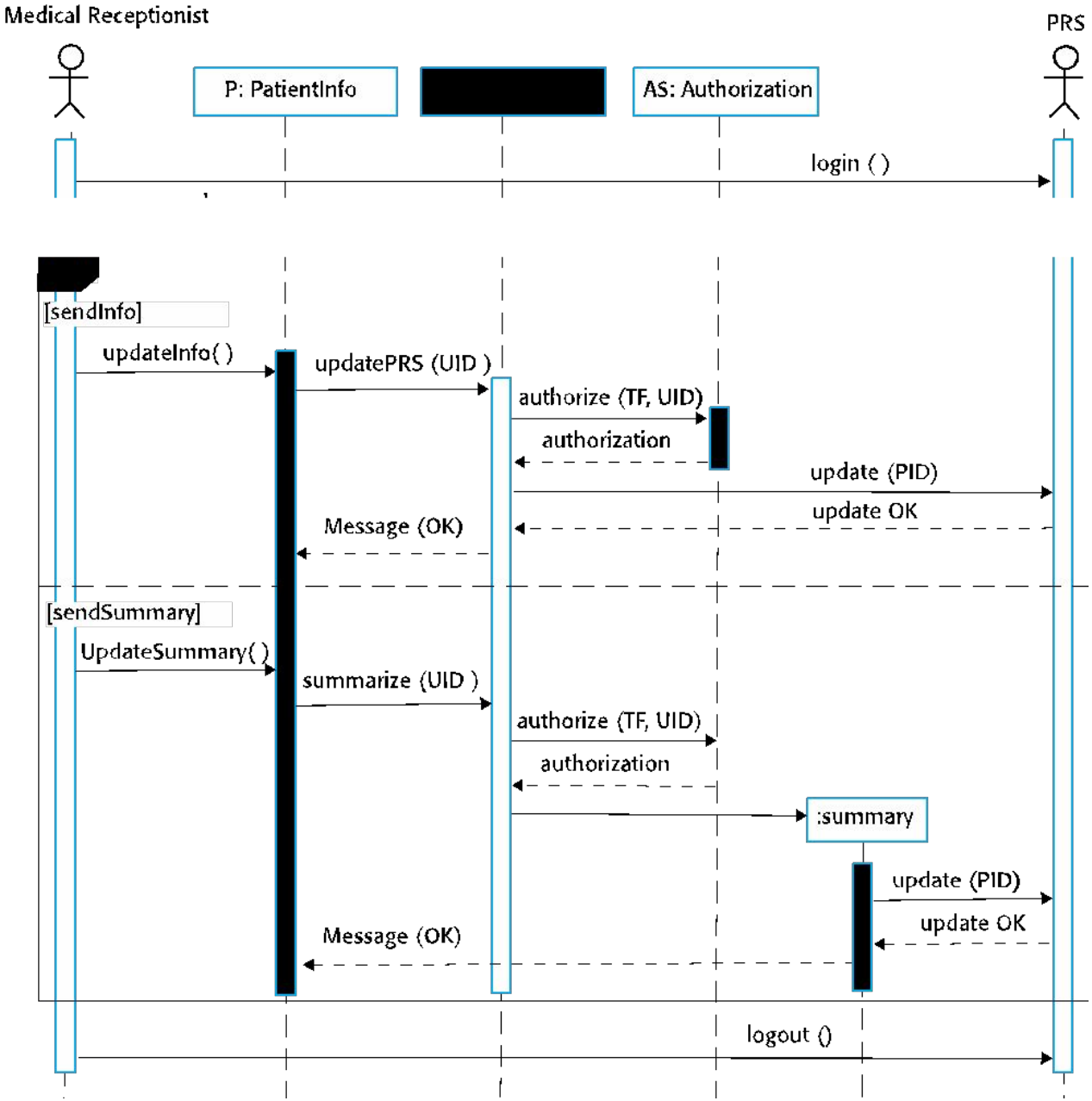
Sequence diagrams

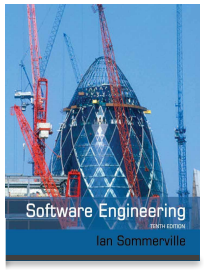


- Sequence diagrams are part of the UML and are used to model the interactions between the actors and the objects within a system.
- A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
- The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
- Interactions between objects are indicated by annotated arrows.

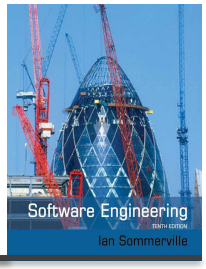
Sequence diagram for View patient information





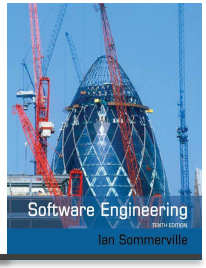


Sequence diagram for Transfer Data



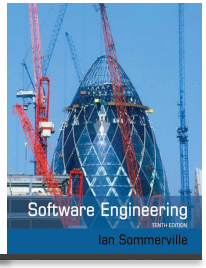
Structural models

Structural models



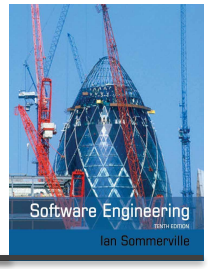
- Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.
- Structural models may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executing.
- You create structural models of a system when you are discussing and designing the system architecture.

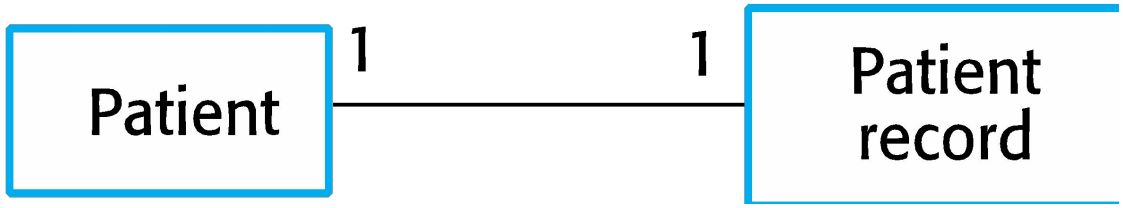
Class diagrams



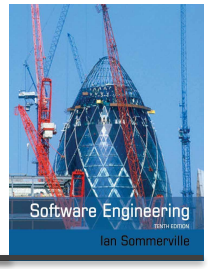
- Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.
- An object class can be thought of as a general definition of one kind of system object.
- An association is a link between classes that indicates that there is some relationship between these classes.
- When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, doctor, etc.

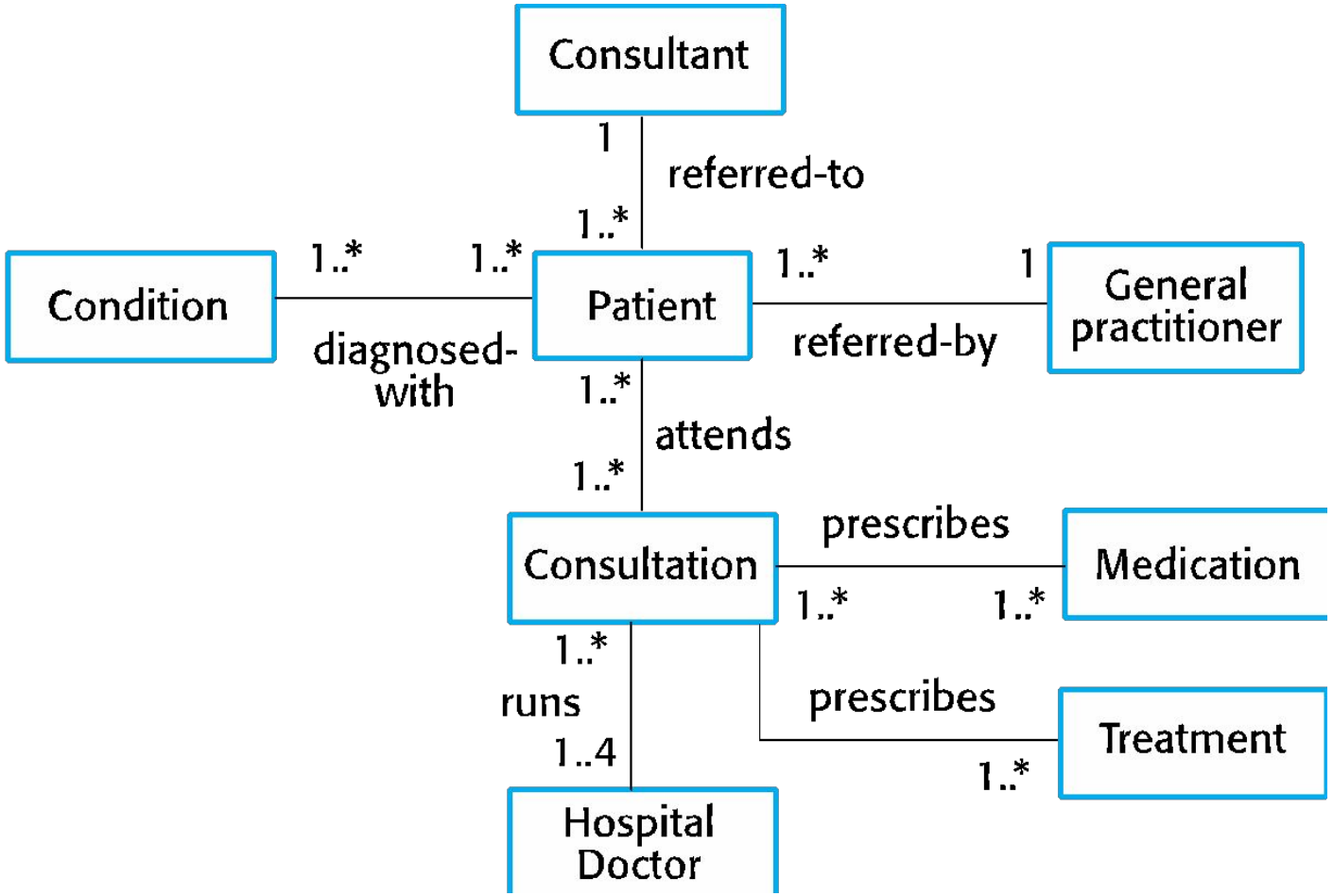
UML classes and association



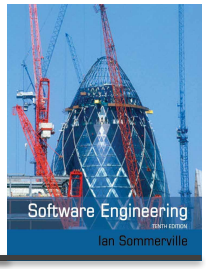


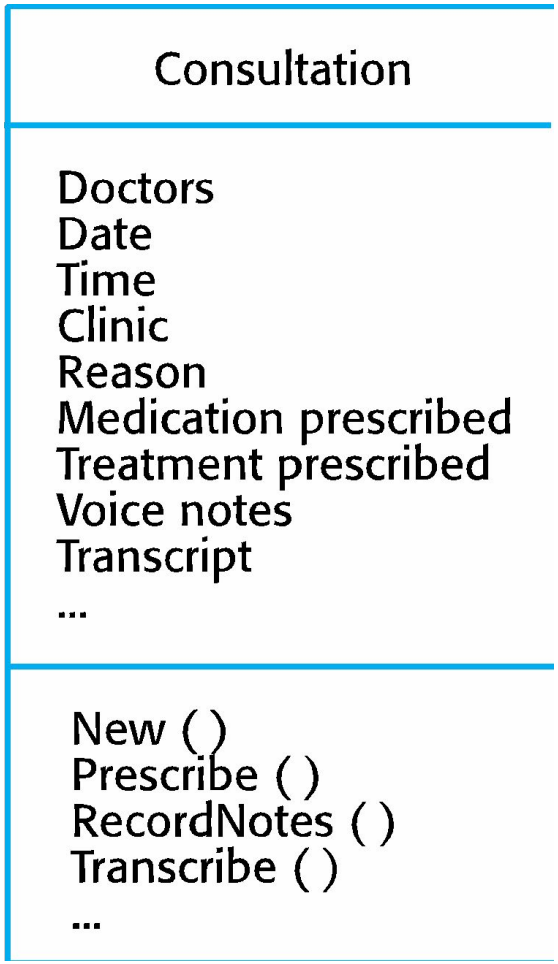
Classes and associations in the MHC-PMS



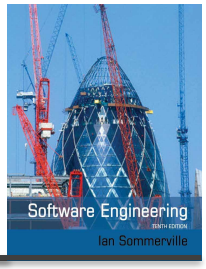


The Consultation class



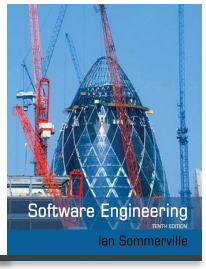


Generalization



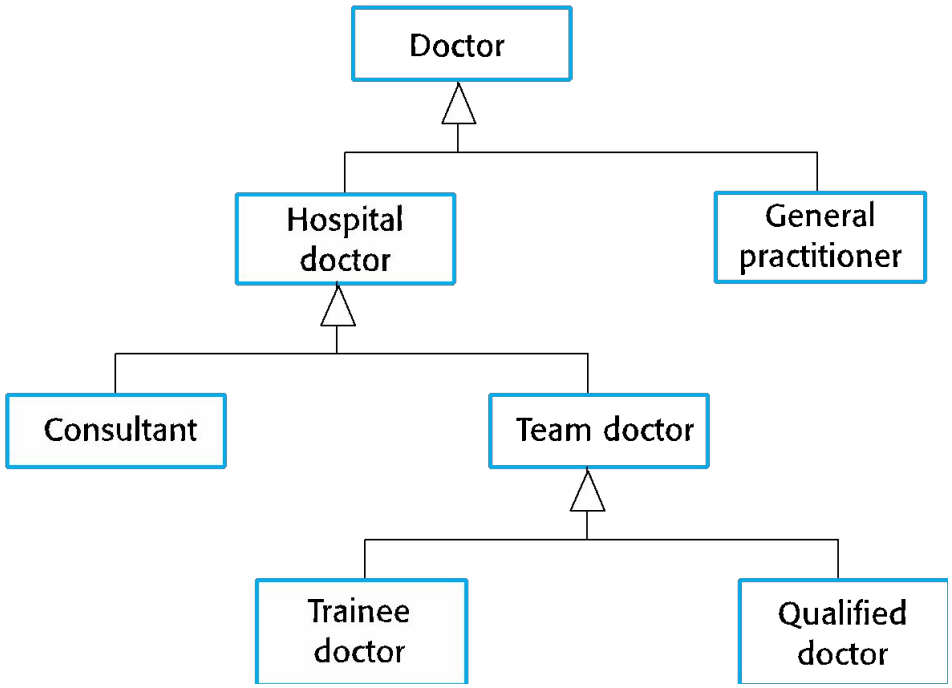
- Generalization is an everyday technique that we use to manage complexity.
- Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of these classes.
- This allows us to infer that different members of these classes have some common characteristics e.g. squirrels and rats are rodents.

Generalization

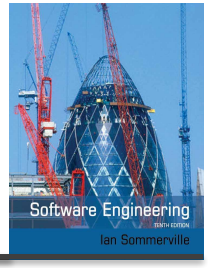


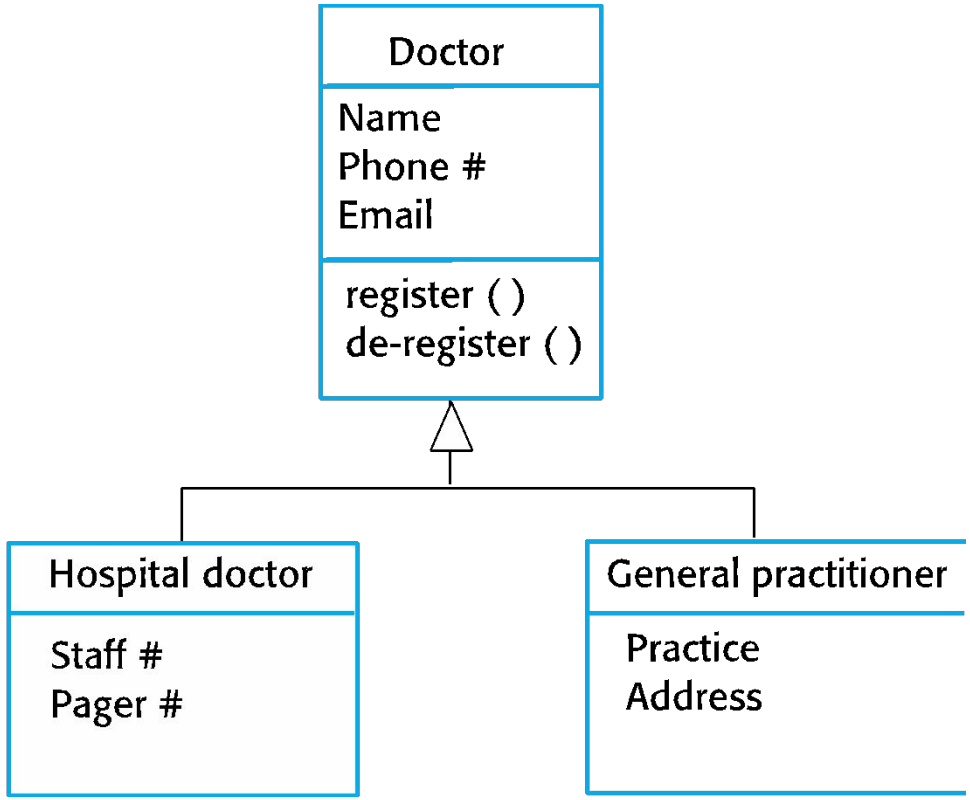
- In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change.
- In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.
- In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes.
- The lower-level classes are subclasses inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.

A generalization hierarchy

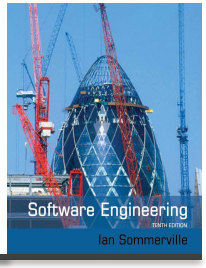


A generalization hierarchy with added detail



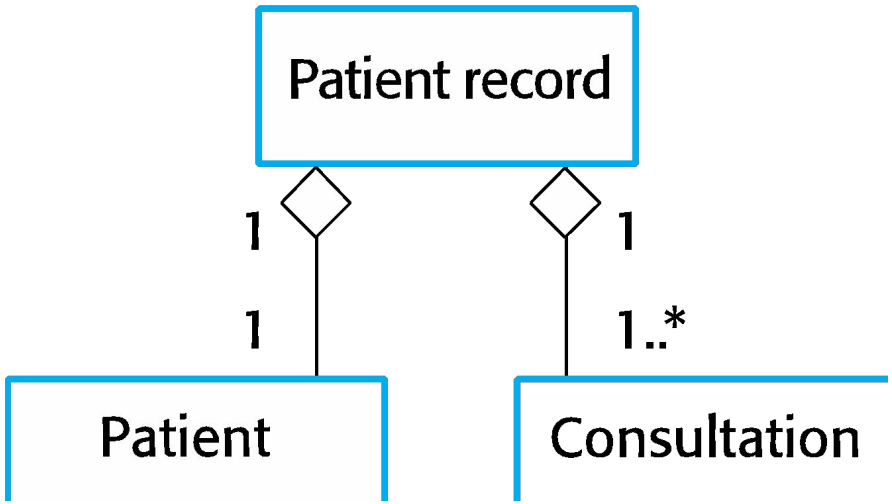


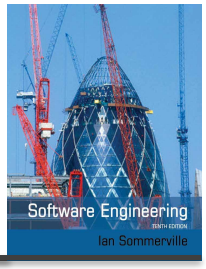
Object class aggregation models



- An aggregation model shows how classes that are collections are composed of other classes.
- Aggregation models are similar to the part-of relationship in semantic data models.

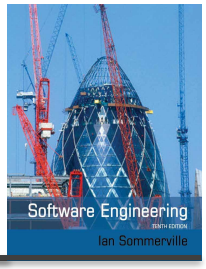
The aggregation association





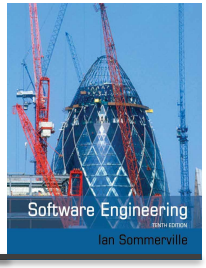
Behavioral models

Behavioral models



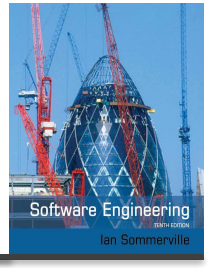
- Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.
- You can think of these stimuli as being of two types:
  - Data Some data arrives that has to be processed by the system.
  - Events Some event happens that triggers system processing. Events may have associated data, although this is not always the case.

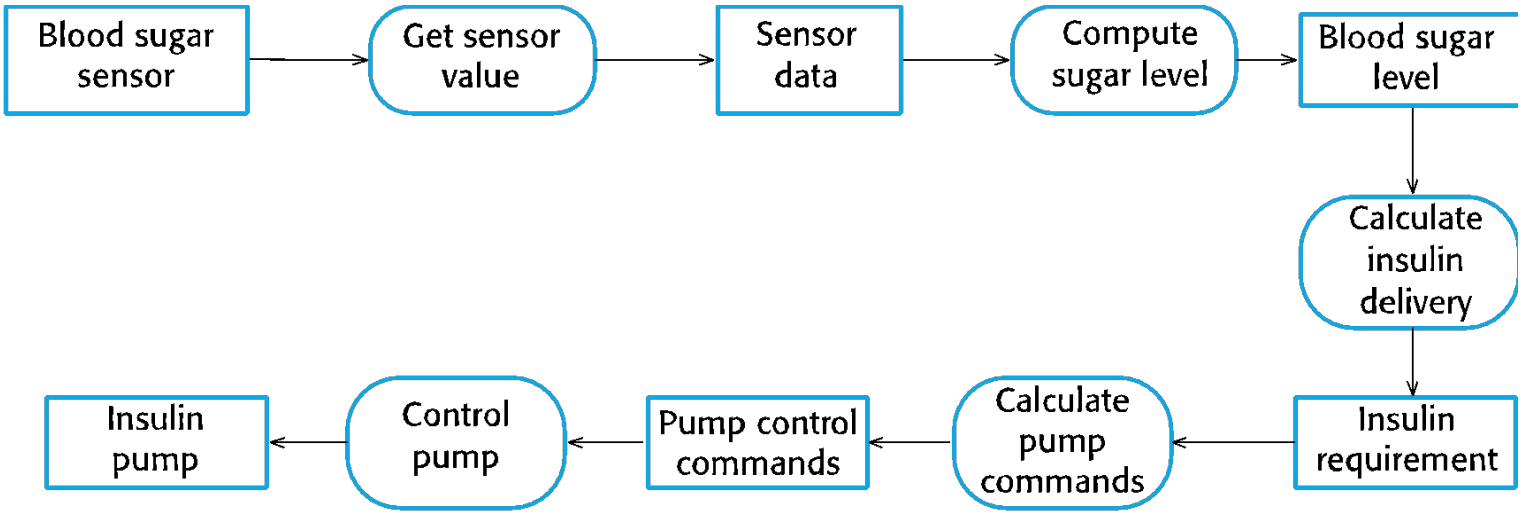
Data-driven modeling



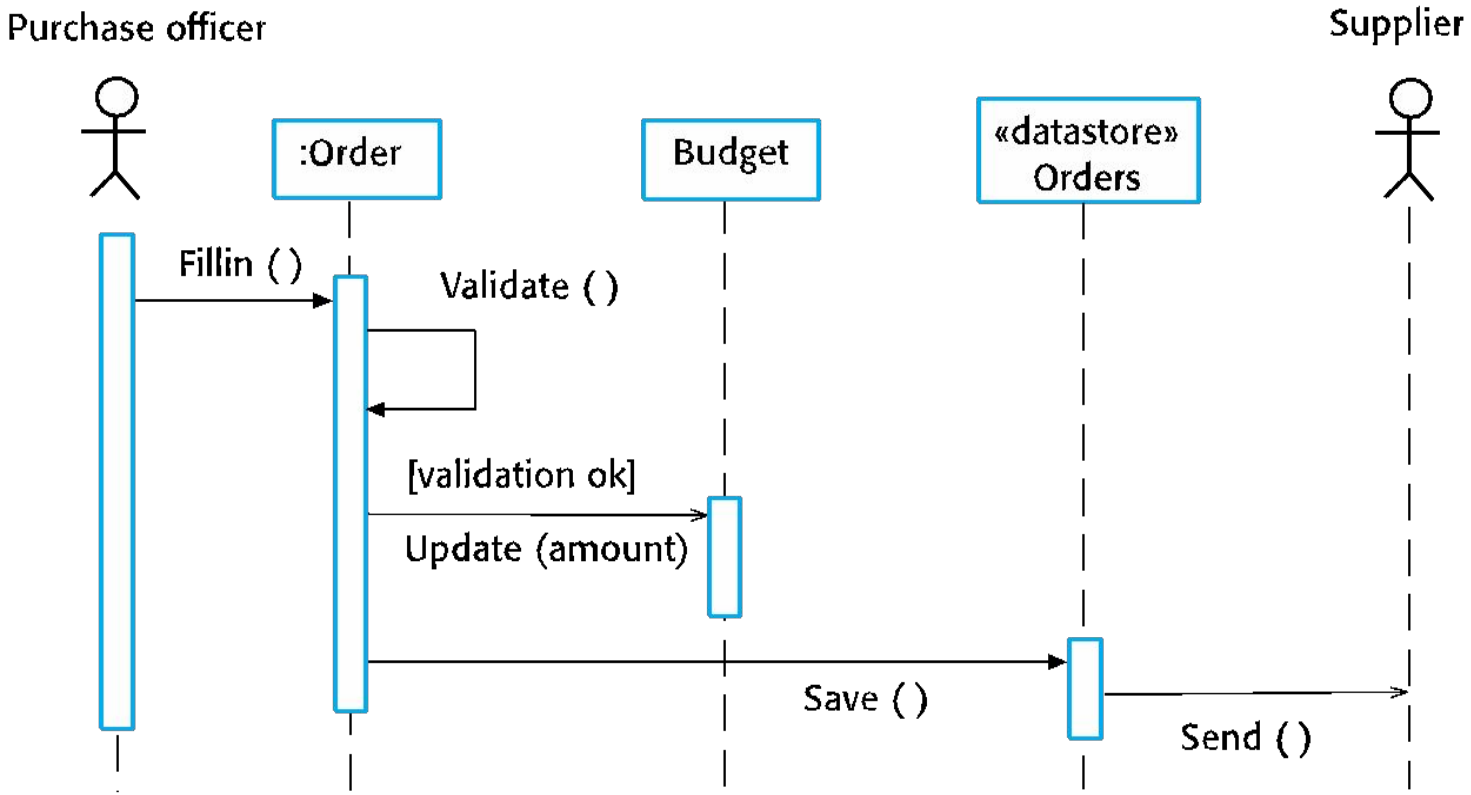
- Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing.
- Data-driven models show the sequence of actions involved in processing input data and generating an associated output.
- They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system.

An activity model of the insulin pump's operation

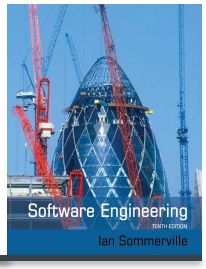




Order processing

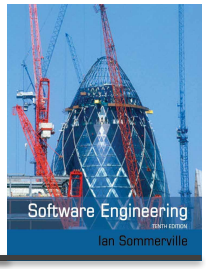


Event-driven modeling



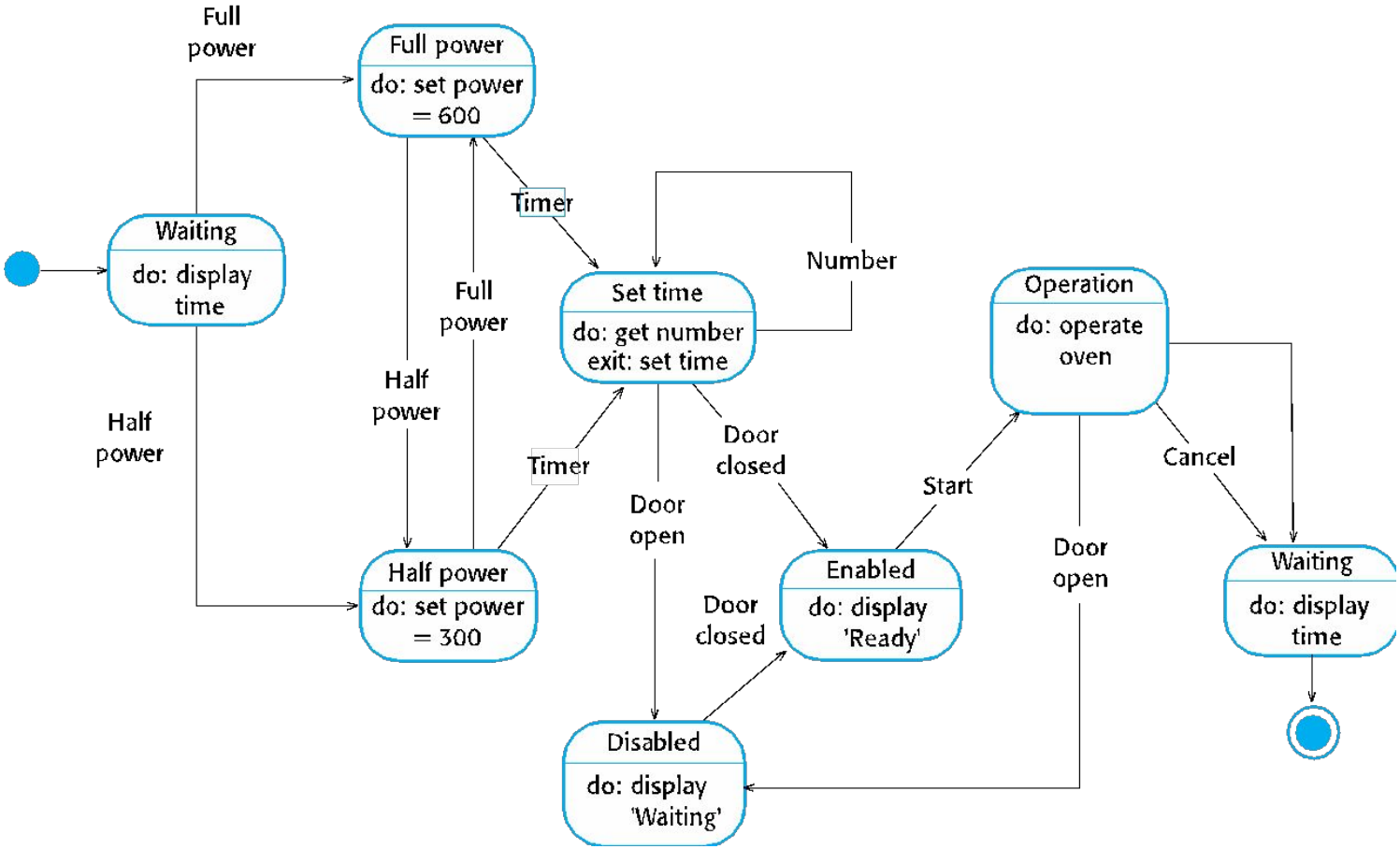
- Real-time systems are often event-driven, with minimal data processing. For example, a landline phone switching system responds to events such as ‘receiver off hook’ by generating a dial tone.
- Event-driven modeling shows how a system responds to external and internal events.
- It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.

State machine models

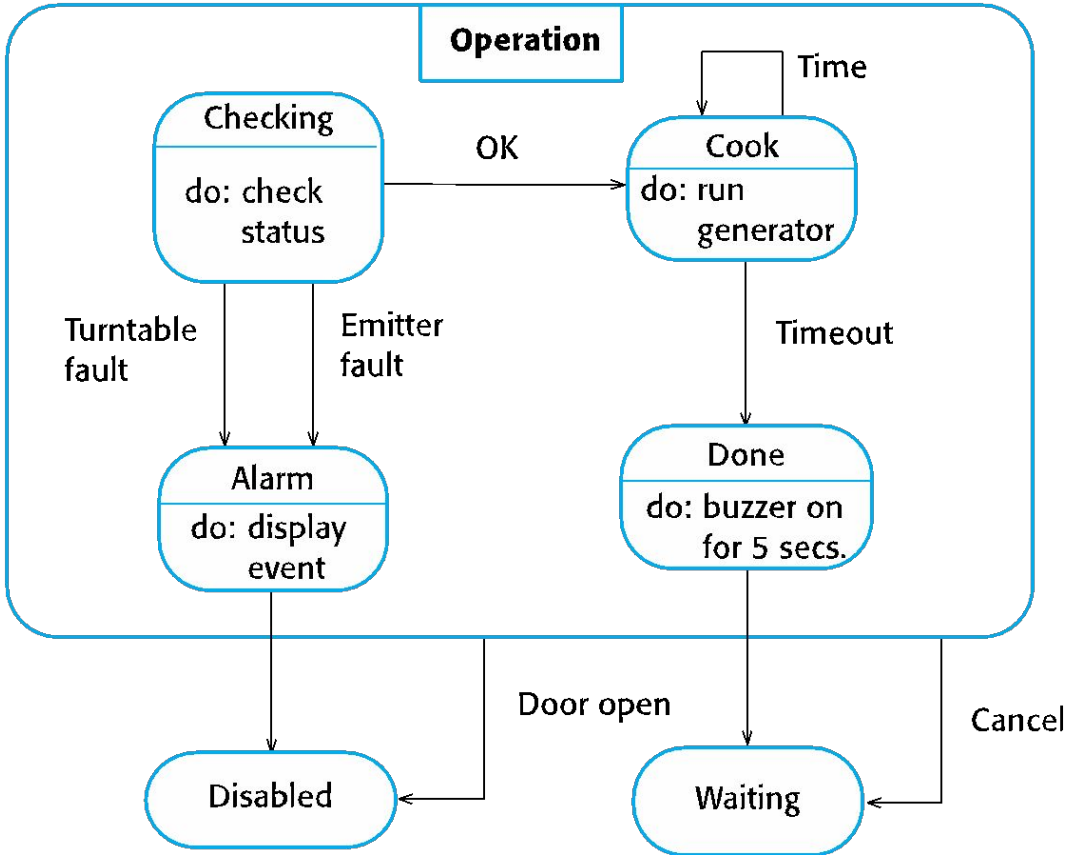


- These model the behaviour of the system in response to external and internal events.
- They show the system's responses to stimuli so are often used for modelling real-time systems.
- State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
- Statecharts are an integral part of the UML and are used to represent state machine models.

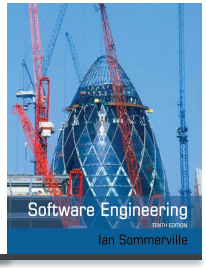
State diagram of a microwave oven



Microwave oven operation

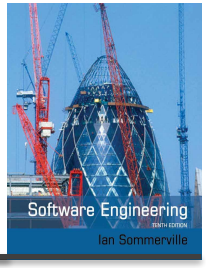


States and stimuli for the microwave oven (a)

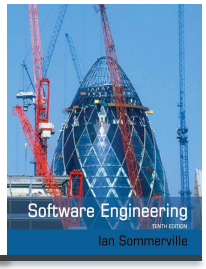


State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows ‘Half power’.
Full power	The oven power is set to 600 watts. The display shows ‘Full power’.
Set time	The cooking time is set to the user’s input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows ‘Not ready’.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows ‘Ready to cook’.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows ‘Cooking complete’ while buzzer is sounding.

States and stimuli for the microwave oven (b)

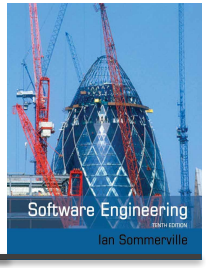


Stimulus	Description
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the Start button.
Cancel	The user has pressed the Cancel button.



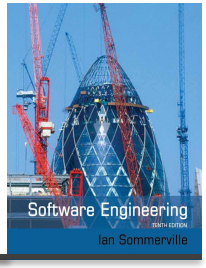
Model-driven engineering

Model-driven engineering



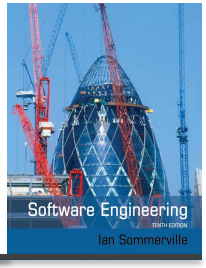
- Model-driven engineering (MDE) is an approach to software development where models rather than programs are the principal outputs of the development process.
- The programs that execute on a hardware/software platform are then generated automatically from the models.
- Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

Usage of model-driven engineering



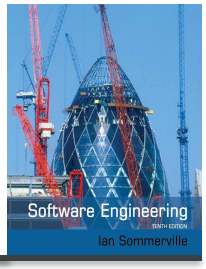
- Model-driven engineering is still at an early stage of development, and it is unclear whether or not it will have a significant effect on software engineering practice.
- Pros
  - Allows systems to be considered at higher levels of abstraction
  - Generating code automatically means that it is cheaper to adapt systems to new platforms.
- Cons
  - Models for abstraction and not necessarily right for implementation.
  - Savings from generating code may be outweighed by the costs of developing translators for new platforms.

Model driven architecture



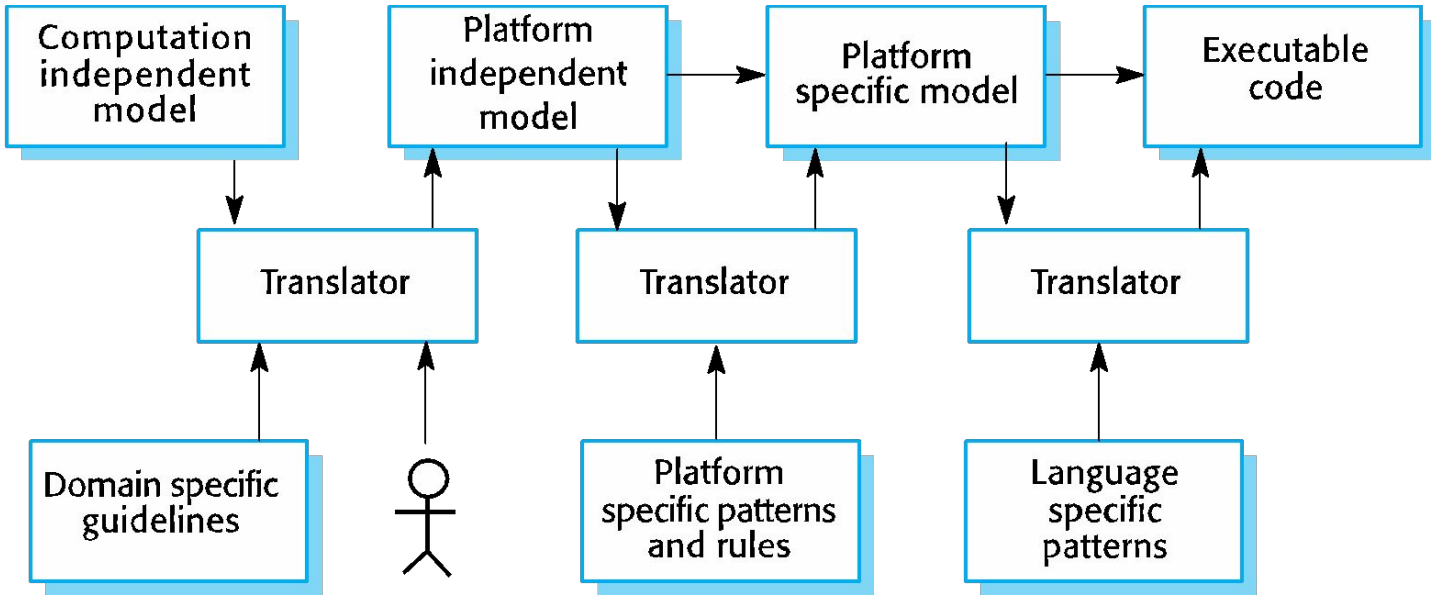
- Model-driven architecture (MDA) was the precursor of more general model-driven engineering
- MDA is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system.
- Models at different levels of abstraction are created. From a high-level, platform independent model, it is possible, in principle, to generate a working program without manual intervention.

Types of model

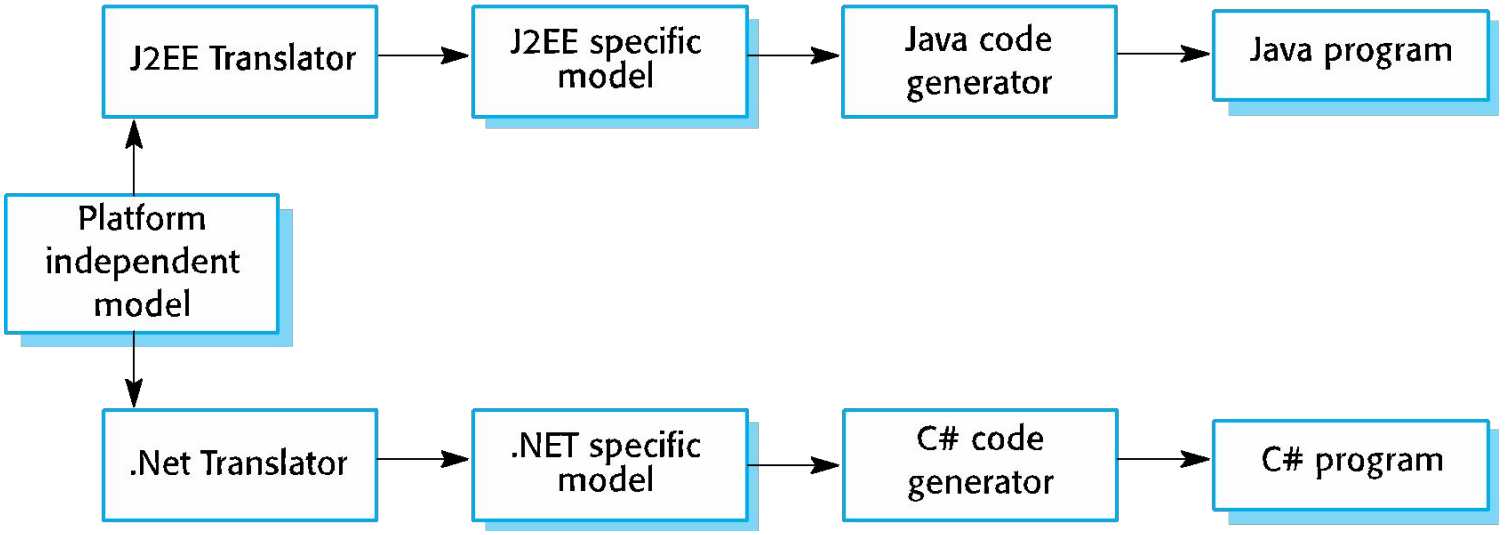


- A computation independent model (CIM)
  - These model the important domain abstractions used in a system. CIMs are sometimes called domain models.
- A platform independent model (PIM)
  - These model the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.
- Platform specific models (PSM)
  - These are transformations of the platform-independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform-specific detail.

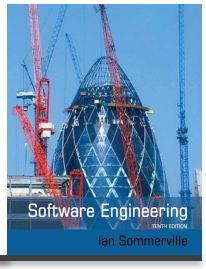
MDA transformations



Multiple platform-specific models

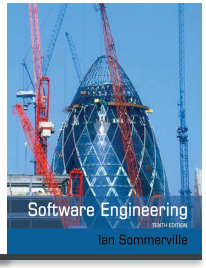


Agile methods and MDA



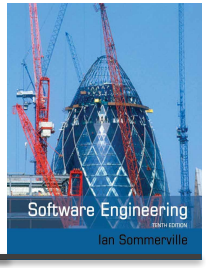
- The developers of MDA claim that it is intended to support an iterative approach to development and so can be used within agile methods.
- The notion of extensive up-front modeling contradicts the fundamental ideas in the agile manifesto and I suspect that few agile developers feel comfortable with model-driven engineering.
- If transformations can be completely automated and a complete program generated from a PIM, then, in principle, MDA could be used in an agile development process as no separate coding would be required.

Adoption of MDA



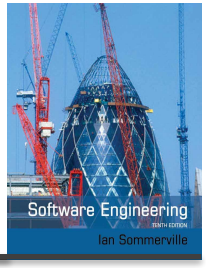
- A range of factors has limited the adoption of MDE/MDA
- Specialized tool support is required to convert models from one level to another
- There is limited tool availability and organizations may require tool adaptation and customisation to their environment
- For the long-lifetime systems developed using MDA, companies are reluctant to develop their own tools or rely on small companies that may go out of business

Adoption of MDA



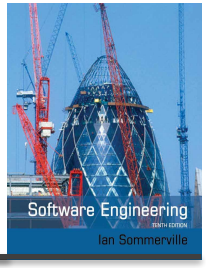
- Models are a good way of facilitating discussions about a software design. Howeverthe abstractions that are useful for discussions may not be the right abstractions for implementation.
- For most complex systems, implementation is not the major problem – requirements engineering, security and dependability, integration with legacy systems and testing are all more significant.

Adoption of MDA



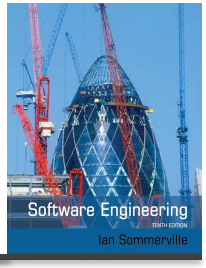
- The arguments for platform-independence are only valid for large, long-lifetime systems. For software products and information systems, the savings from the use of MDA are likely to be outweighed by the costs of its introduction and tooling.
- The widespread adoption of agile methods over the same period that MDA was evolving has diverted attention away from model-driven approaches.

Key points

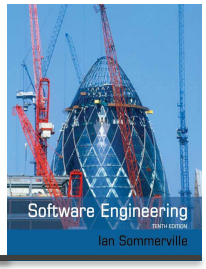


- A model is an abstract view of a system that ignores system details. Complementary system models can be developed to show the system's context, interactions, structure and behavior.
- Context models show how a system that is being modeled is positioned in an environment with other systems and processes.
- Use case diagrams and sequence diagrams are used to describe the interactions between users and systems in the system being designed. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.
- Structural models show the organization and architecture of a system. Class diagrams are used to define the static structure of classes in a system and their associations.

Key points

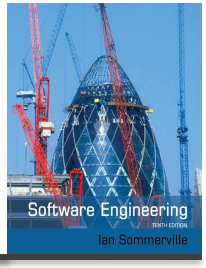


- Behavioral models are used to describe the dynamic behavior of an executing system. This behavior can be modeled from the perspective of the data processed by the system, or by the events that stimulate responses from a system.
- Activity diagrams may be used to model the processing of data, where each activity represents one process step.
- State diagrams are used to model a system's behavior in response to internal or external events.
- Model-driven engineering is an approach to software development in which a system is represented as a set of models that can be automatically transformed to executable code.



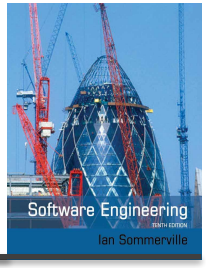
Chapter 6 – Architectural Design

Topics covered



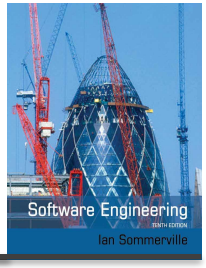
- Architectural design decisions
- Architectural views
- Architectural patterns
- Application architectures

Architectural design



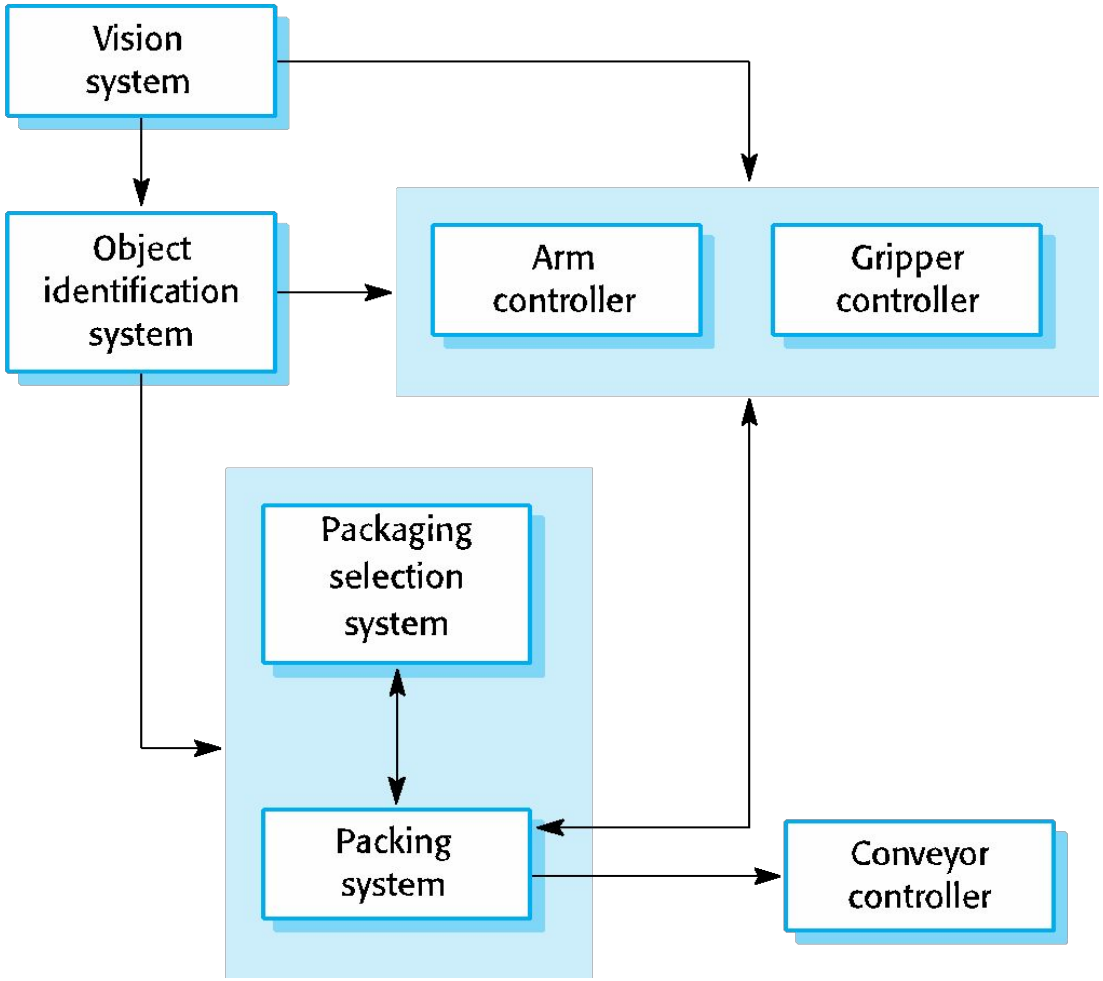
- Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system.
- Architectural design is the critical link between design and requirements engineering, as it identifies the main structural components in a system and 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

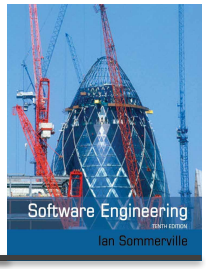


- It is generally accepted that an early stage of agile processes is to design an overall systems architecture.
- Refactoring the system architecture is usually expensive because it affects so many components in the system

The architecture of a packing robot control system

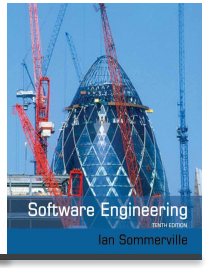


Architectural abstraction



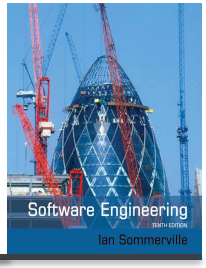
- Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- Architecture in the large is concerned with the architecture of complex enterprise systems that include other 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



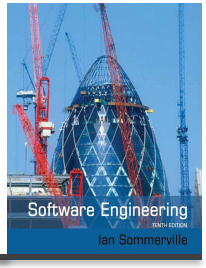
- Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders.
- System analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible.
- Large-scale reuse
  - The architecture may be reusable across a range of systems
  - Product-line architectures may be developed.

Architectural representations



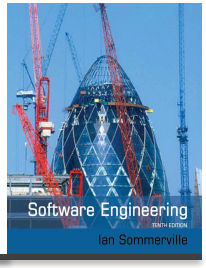
- Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures.
- But these have been criticised because they lack semantics, do not show the types of relationships between entities nor the visible properties of entities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

Box and line diagrams

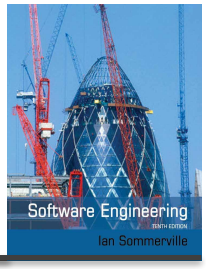


- Very abstract - they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
- However, useful for communication with stakeholders and for project planning.

Use of architectural models

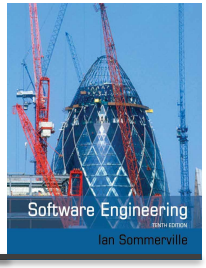


- As a way of facilitating discussion about the system design
  - A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system.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



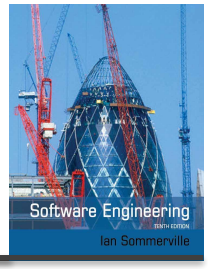
Architectural design decisions

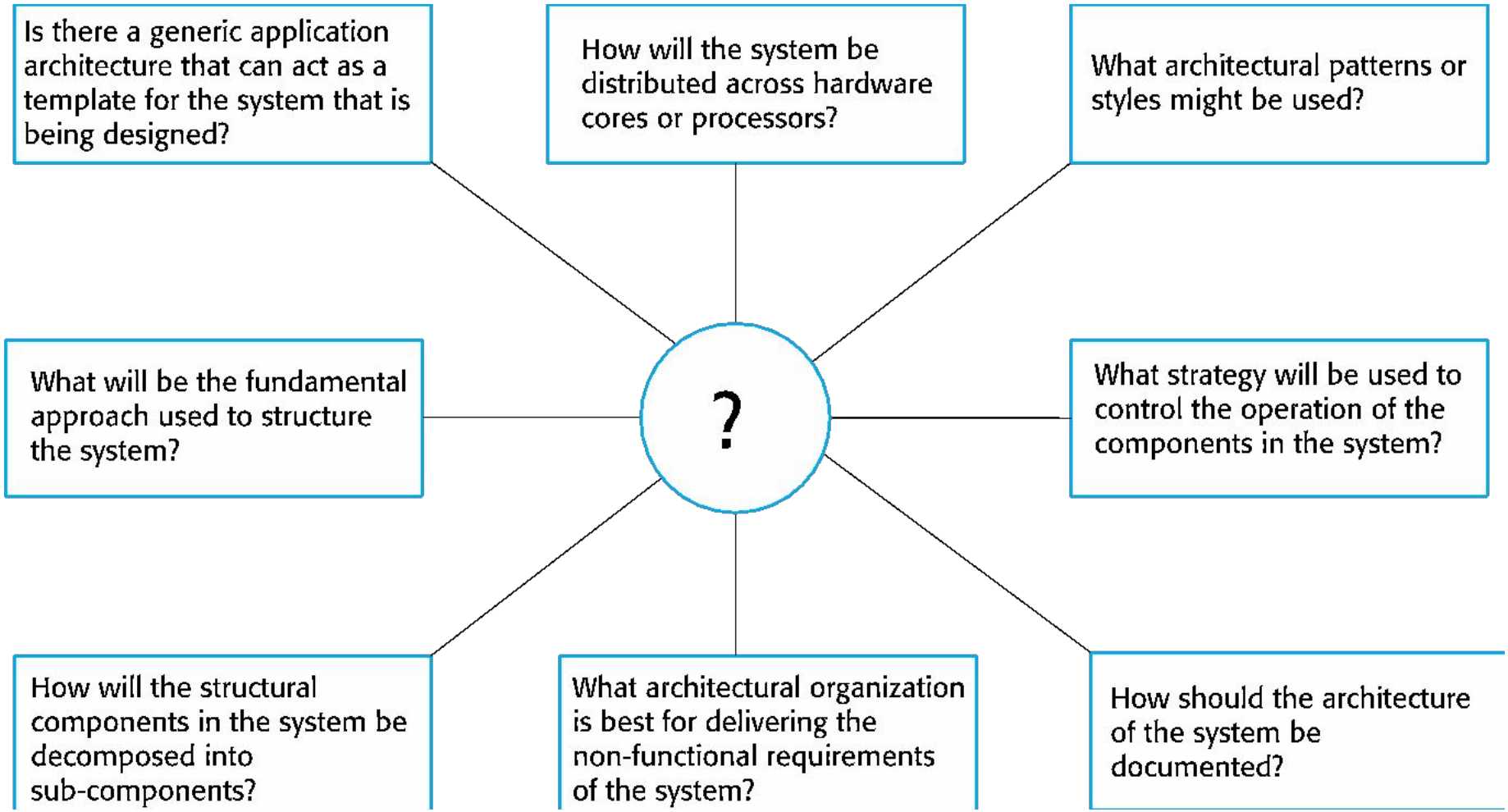
Architectural design decisions



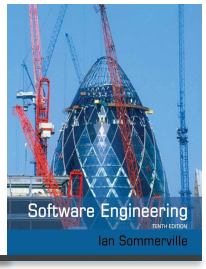
- Architectural design is a creative process so the process differs depending on the type of system being developed.
- However, a number of common decisions span all design processes and these decisions affect the non-functional characteristics of the system.

Architectural design decisions



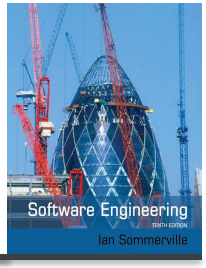


Architecture reuse

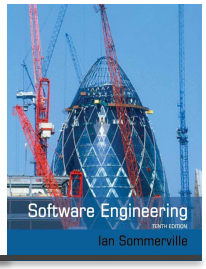


- Systems in the same domain often have similar architectures that reflect domain concepts.
- Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
- The architecture of a system may be designed around one of more architectural patterns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

Architecture and system characteristics

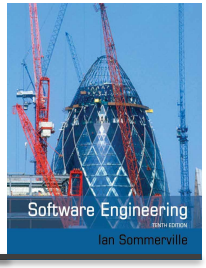


- Performance
  - Localise critical operations and minimise communications. Use large rather than fine-grain components.
- Security
  - Use a layered architecture with critical assets in the inner layers.
- Safety
  - Localise safety-critical features in a small number of sub-systems.
- Availability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



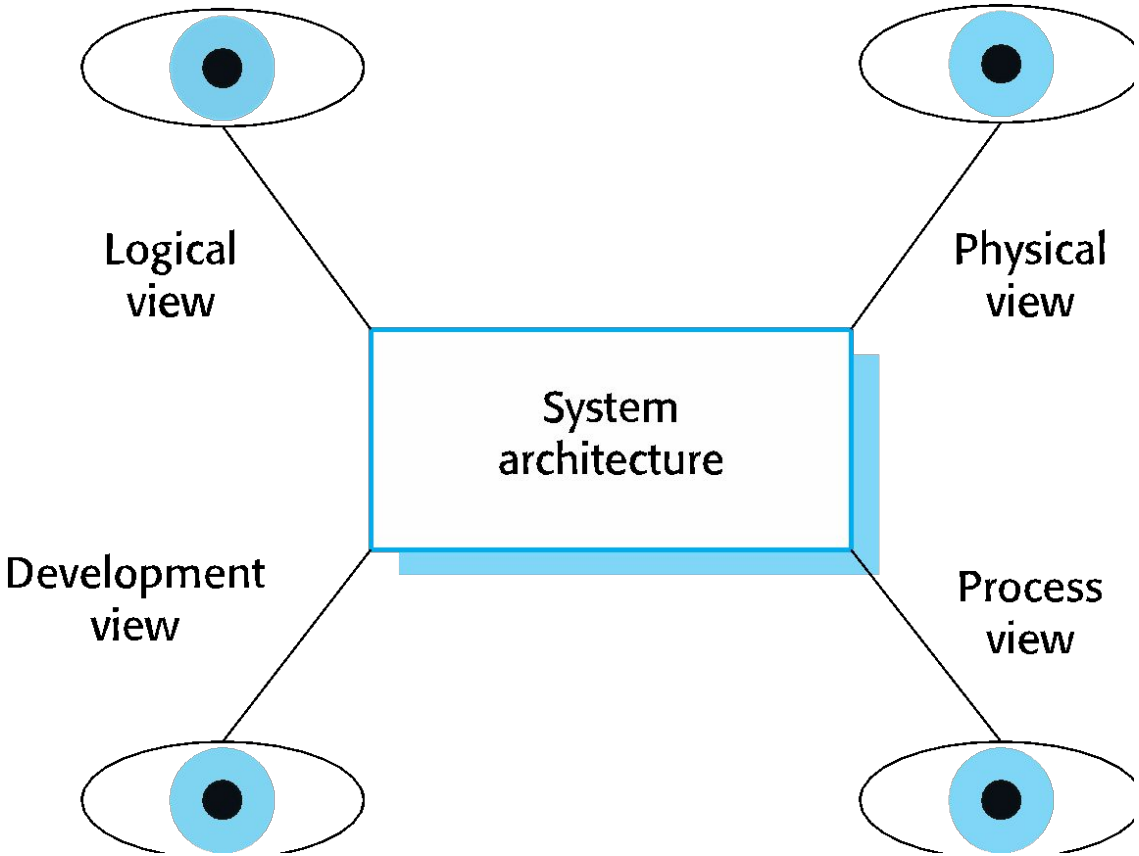
Architectural views

Architectural views

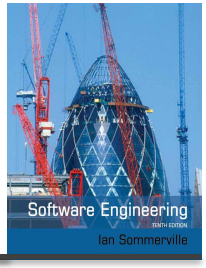


- What views or perspectives are useful when designing and documenting a system's architecture?
- What notations should be used for describing architectural models?
- Each architectural model only shows one view or perspective of the system.
  - It might show how a system is decomposed into modules, 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

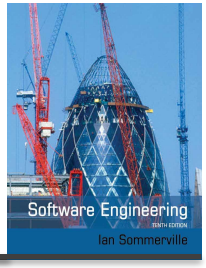


4 + 1 view model of software architecture

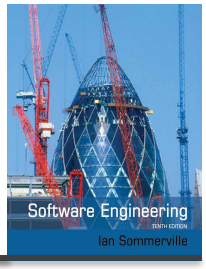


- A logical view, which shows the key abstractions in the system as objects or object classes.
- A process view, which shows how, at run-time, the system is composed of interacting processes.
- A development view, which shows how the software is decomposed for development.
- A physical view, which shows 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

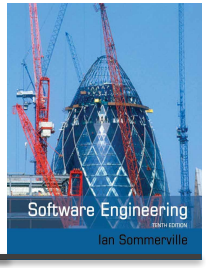


- Some people argue that the Unified Modeling Language (UML) is an appropriate notation for describing and documenting system architectures
- I disagree with this as I do not think that the UML includes abstractions appropriate for high-level system description.
- Architectural description languages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



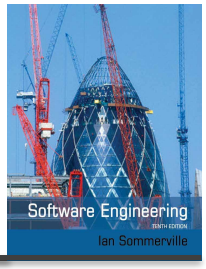
Architectural patterns

Architectural patterns



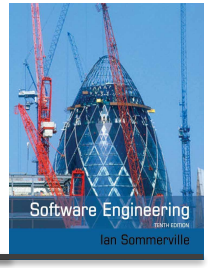
- Patterns are a means of representing, sharing and reusing knowledge.
- An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- Patterns should include information about when they are and when the are not useful.
- Patterns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

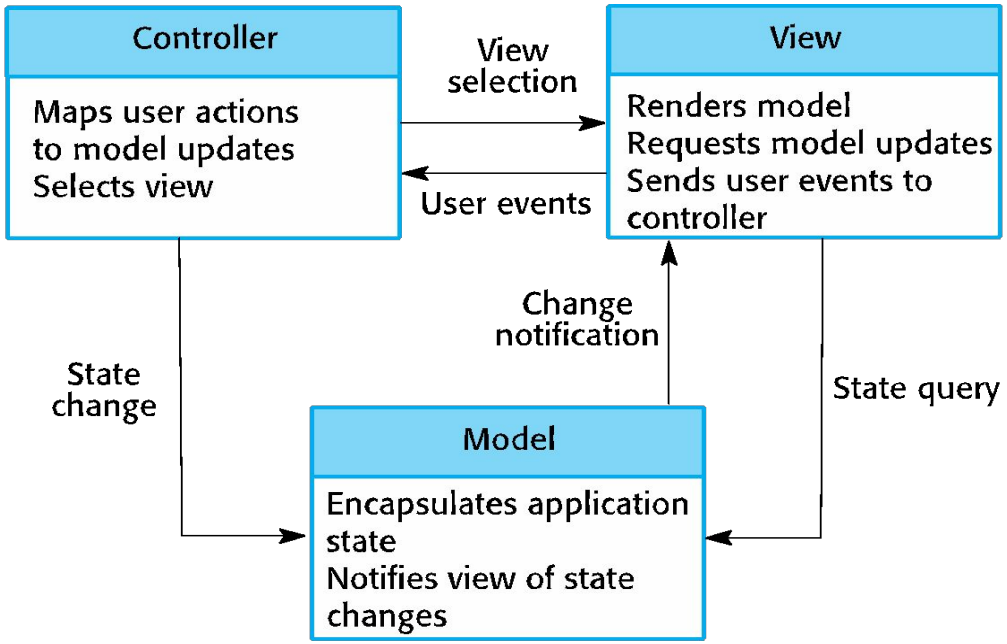
The Model-View-Controller (MVC) pattern



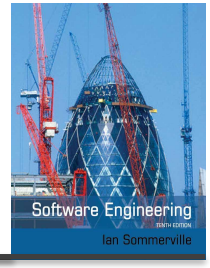
Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines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

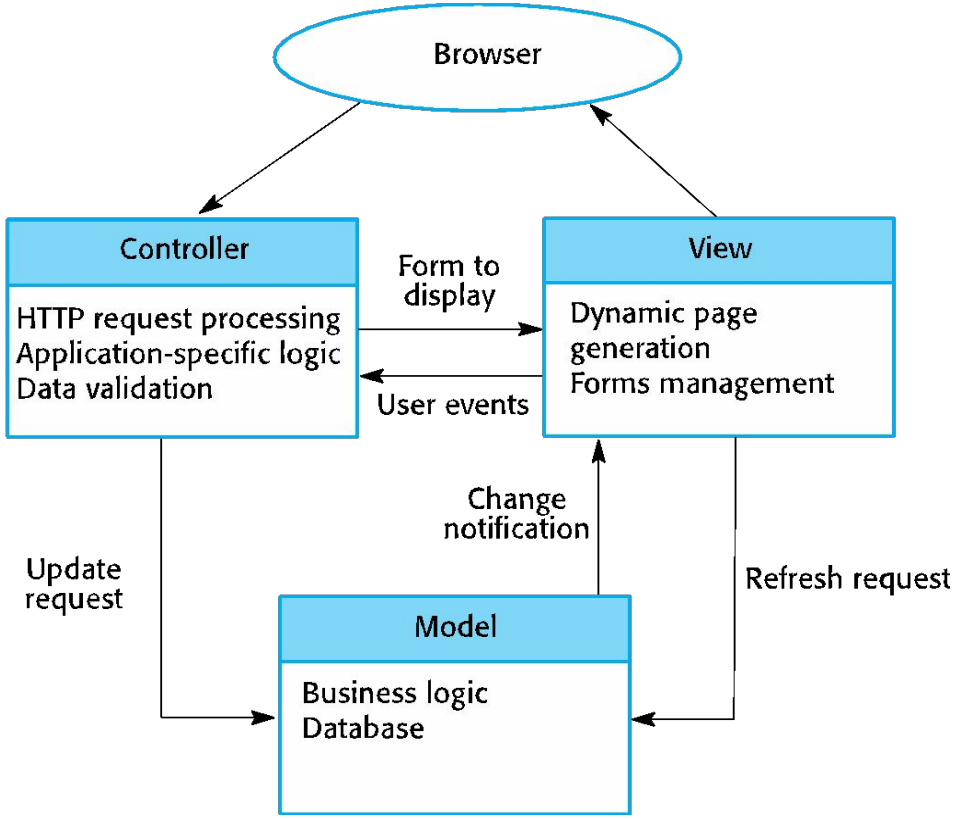
The organization of the Model-View-Controller



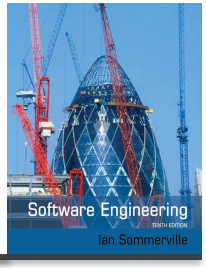


Web application architecture using the MVC pattern



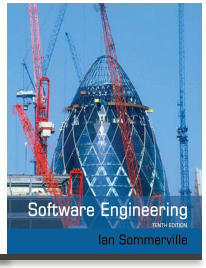


Layered architecture



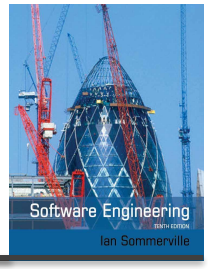
- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
-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

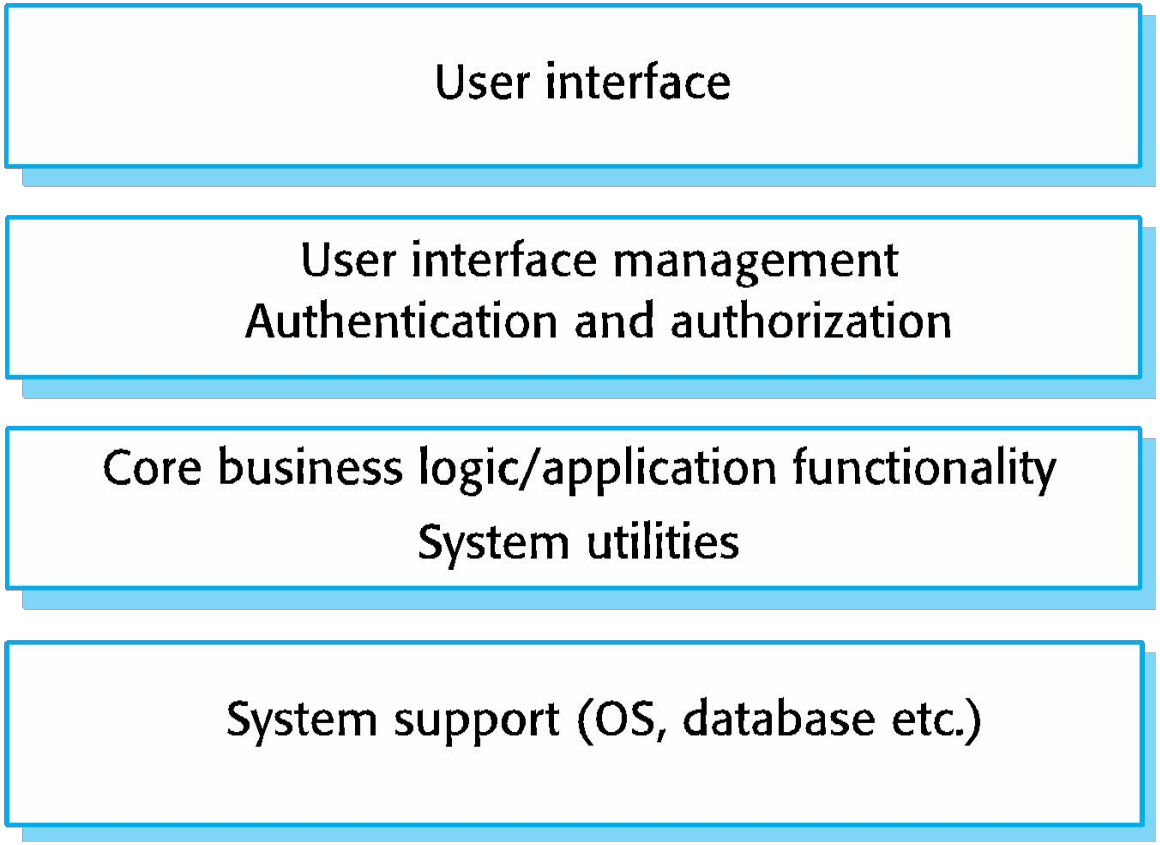
The Layered architecture pattern



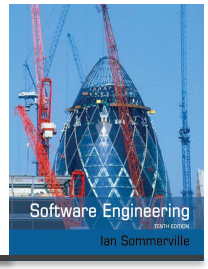
Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A 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.

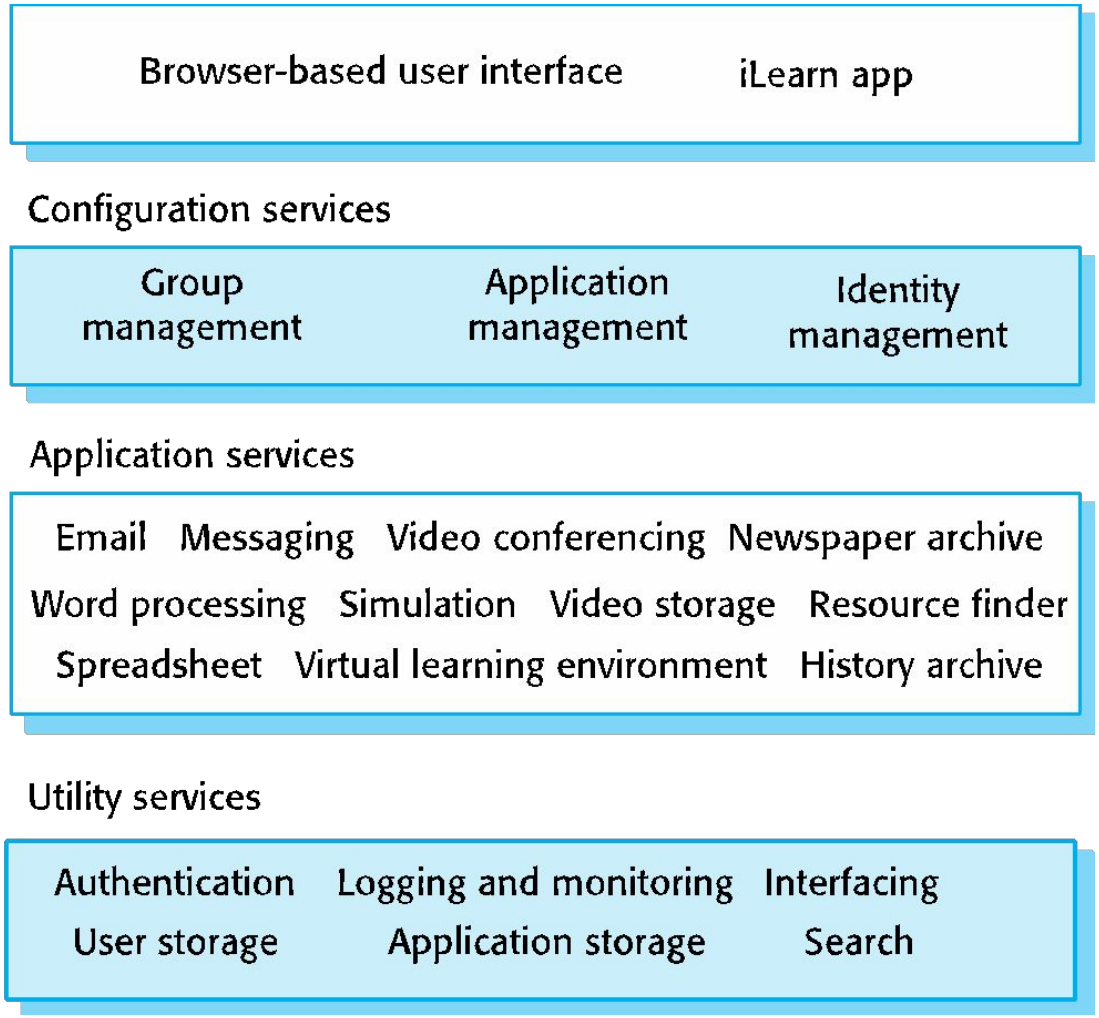
A generic layered architecture



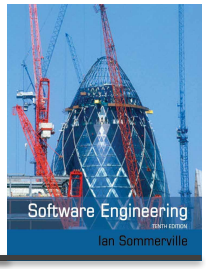


The architecture of the iLearn system



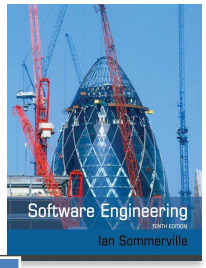


Repository architecture



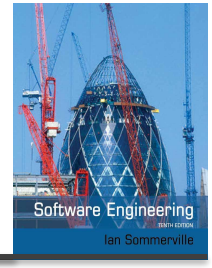
- Sub-systems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared,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

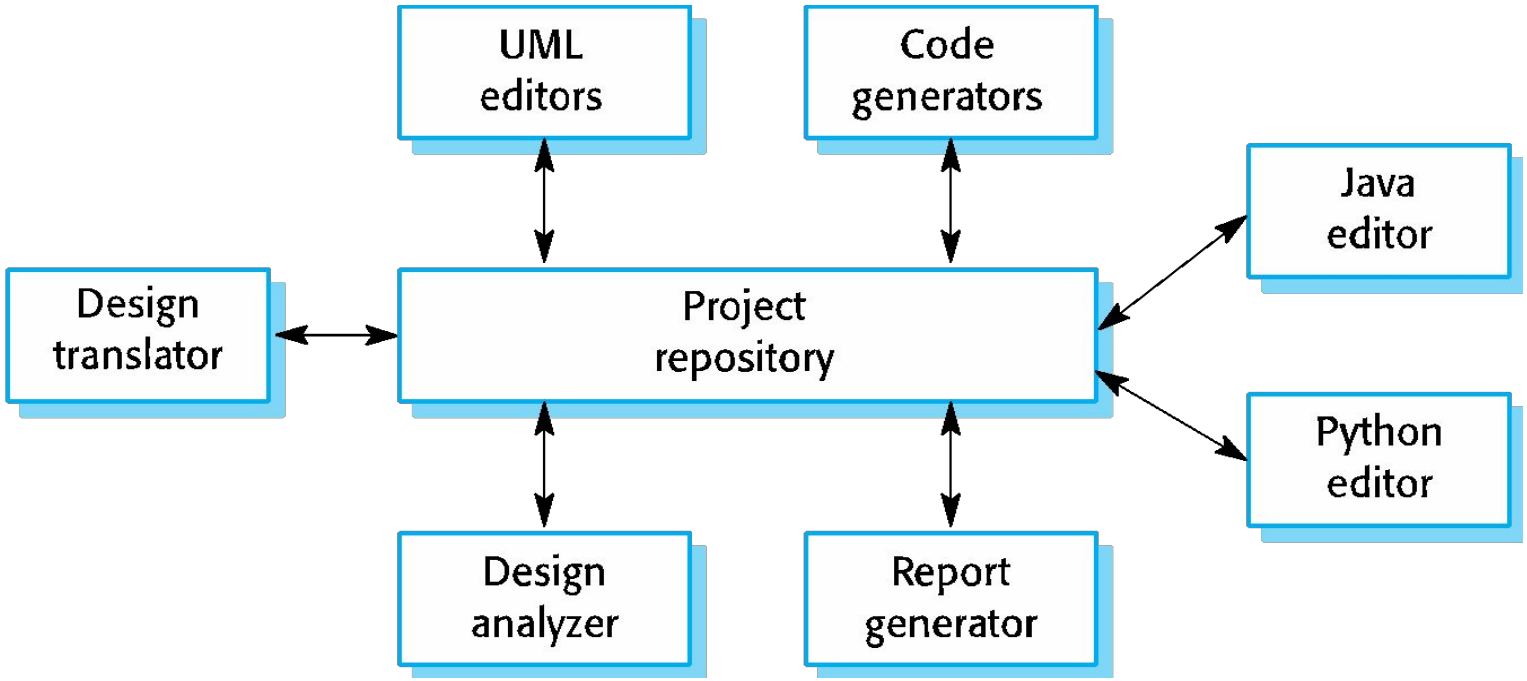
The Repository pattern



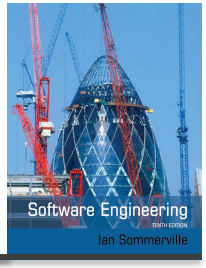
Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information.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



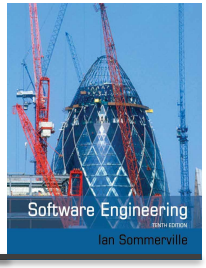


Client-server architecture



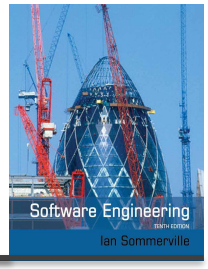
- Distributed system model which shows how data and processing is distributed across a range of components.
  - Can be implemented on a single computer.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.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

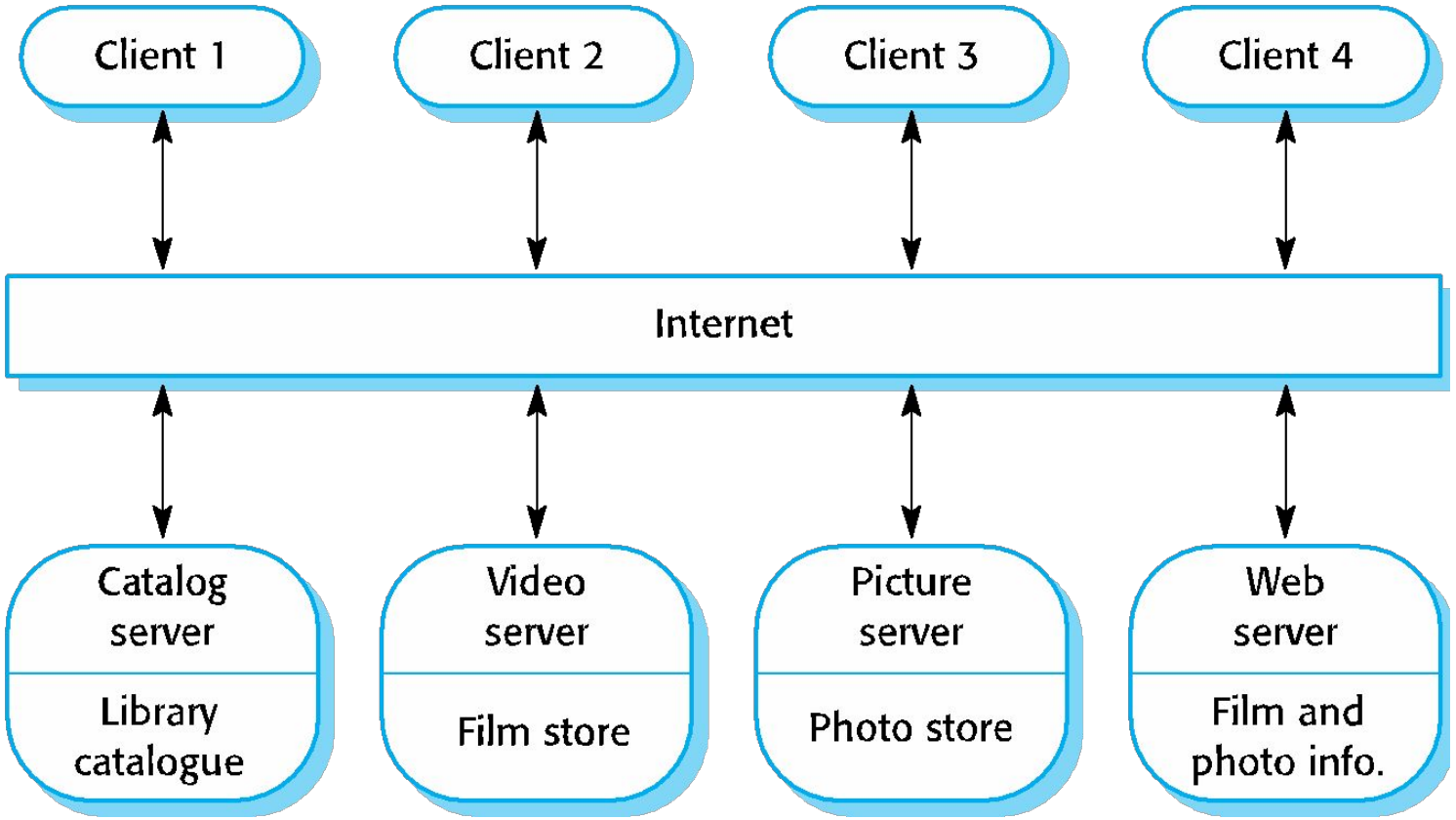
The Client–server pattern



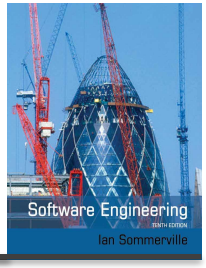
Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD 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



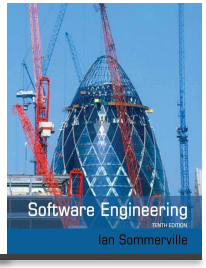


Pipe and filter architecture



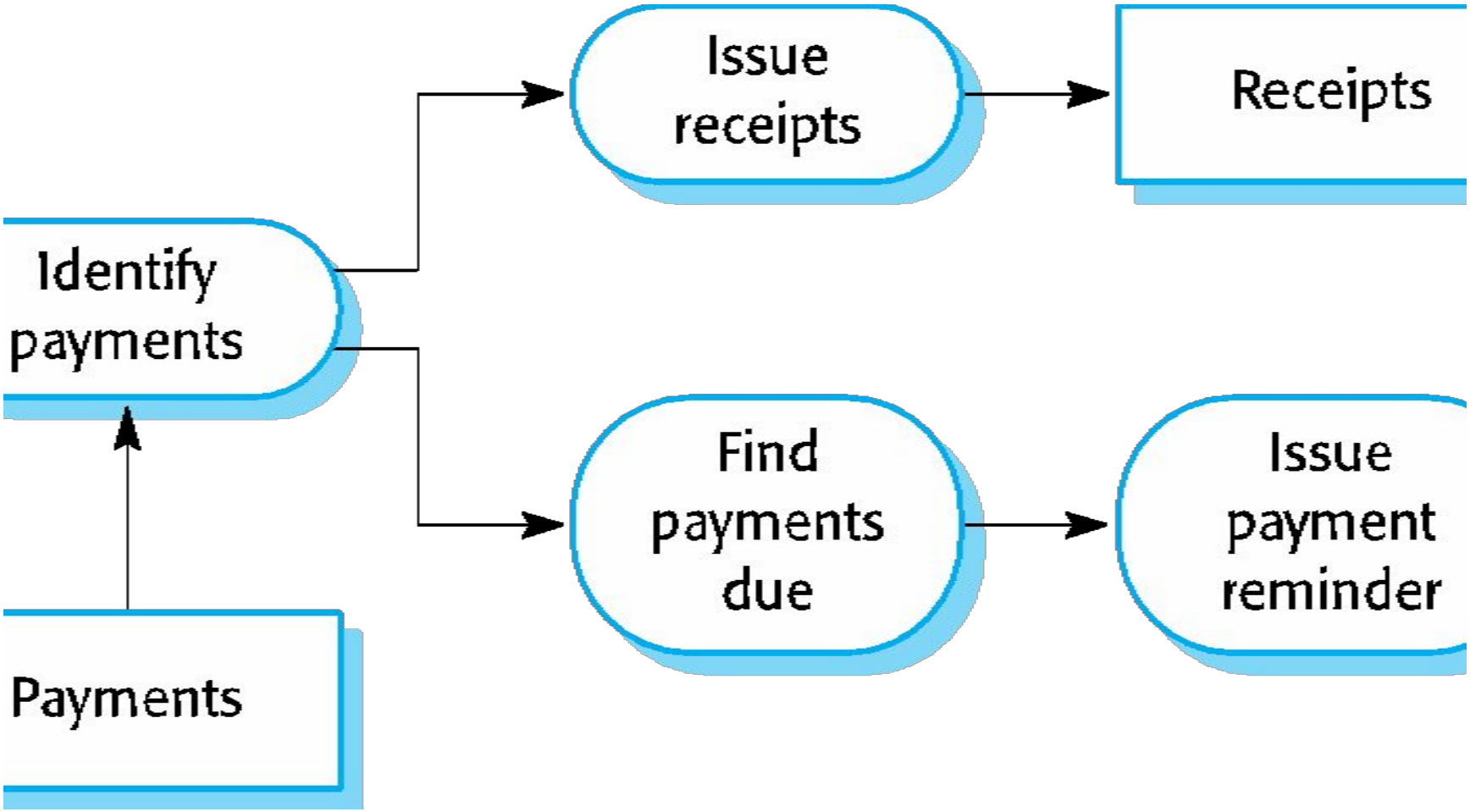
- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.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

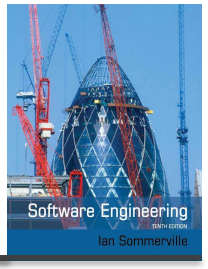
The pipe and filter pattern



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.

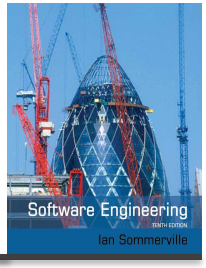
An example of the pipe and filter architecture used in a payments system





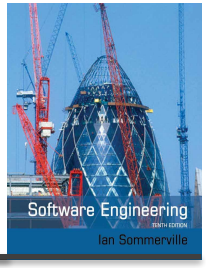
Application architectures

Application architectures



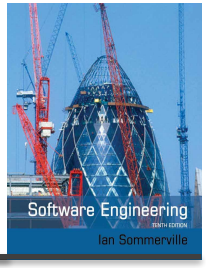
- Application systems are designed to meet an organisational need.
- As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements.
- A generic application architecture is an architecture for a type of software system that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

Use of application architectures



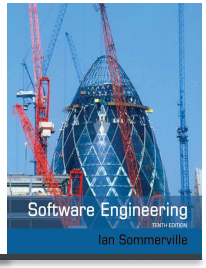
- As a starting point for architectural design.
- As a design checklist.
- As a way of organising the work of the development team.
- As a means of assessing components for reuse.
- As a vocabulary for talking about application types.

Examples of application types



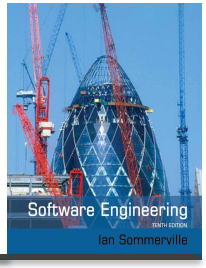
- Data processing applications
  - Data driven applications that process data in batches without explicit user intervention during the processing.
- Transaction processing applications
  - Data-centred applications that process user requests and update information in a system database.
- Event processing 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



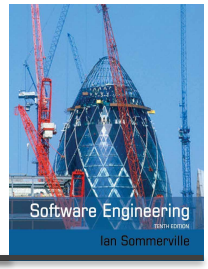
- Two very widely used generic application architectures are transaction processing systems and language processing systems.
- Transaction processing systems
  - E-commerce systems;
  - Reservation systems.
- Language processing systems
  - Compilers;
  - Command interpreters.

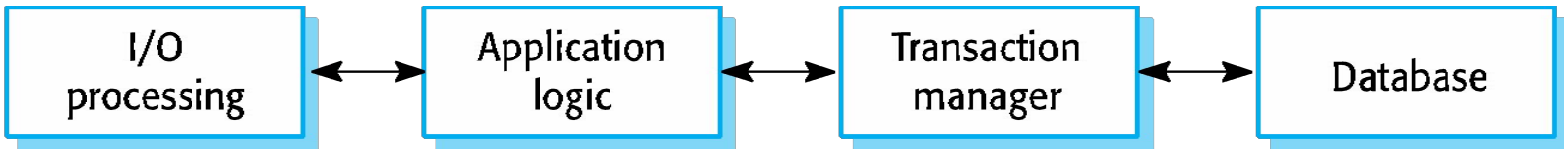
Transaction processing systems



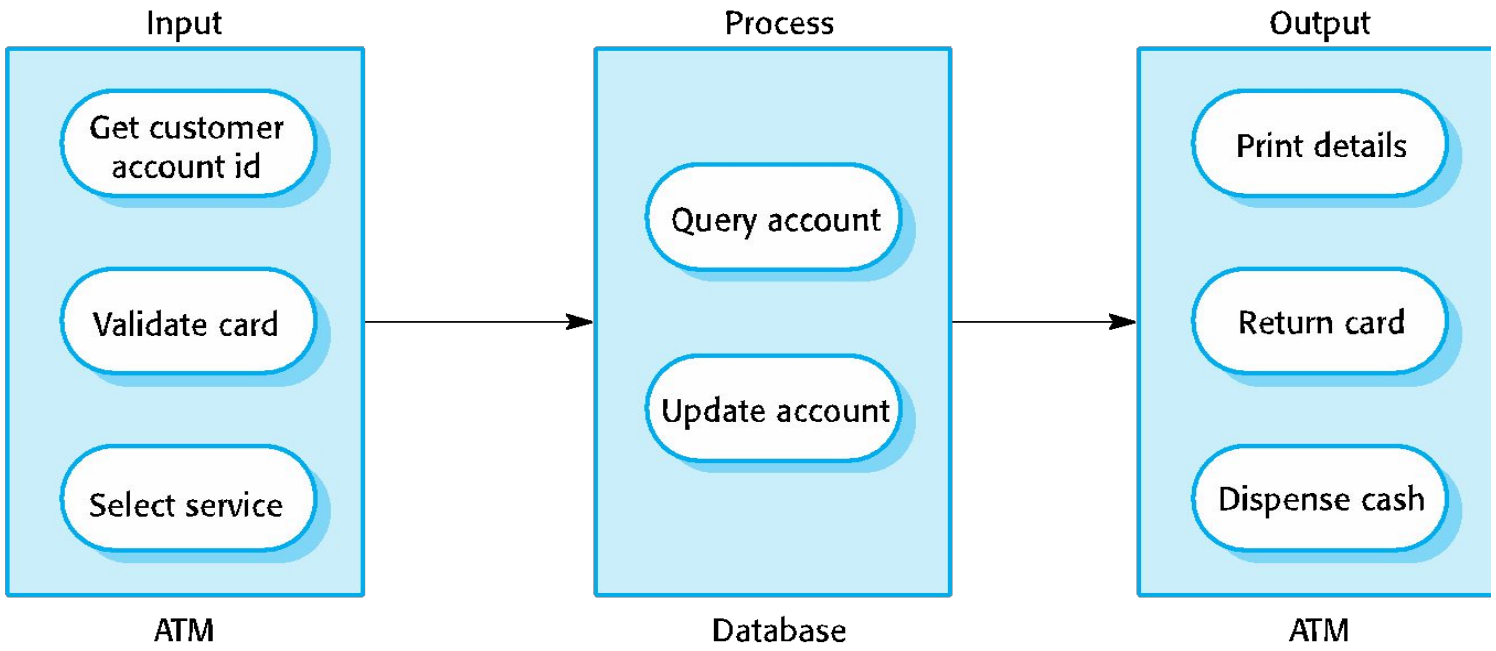
- Process user requests for information from a database or requests to update the database.
- From a user perspective a transaction is:
  - Any coherent sequence of operations that satisfies a goal;
  - For example - find the times of flights from London to Paris.
- Users make asynchronous requests for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

The structure of transaction processing applications

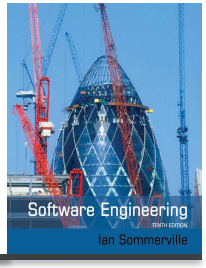




The software architecture of an ATM system

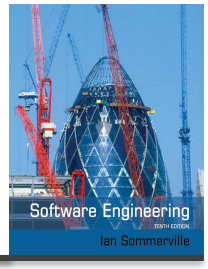


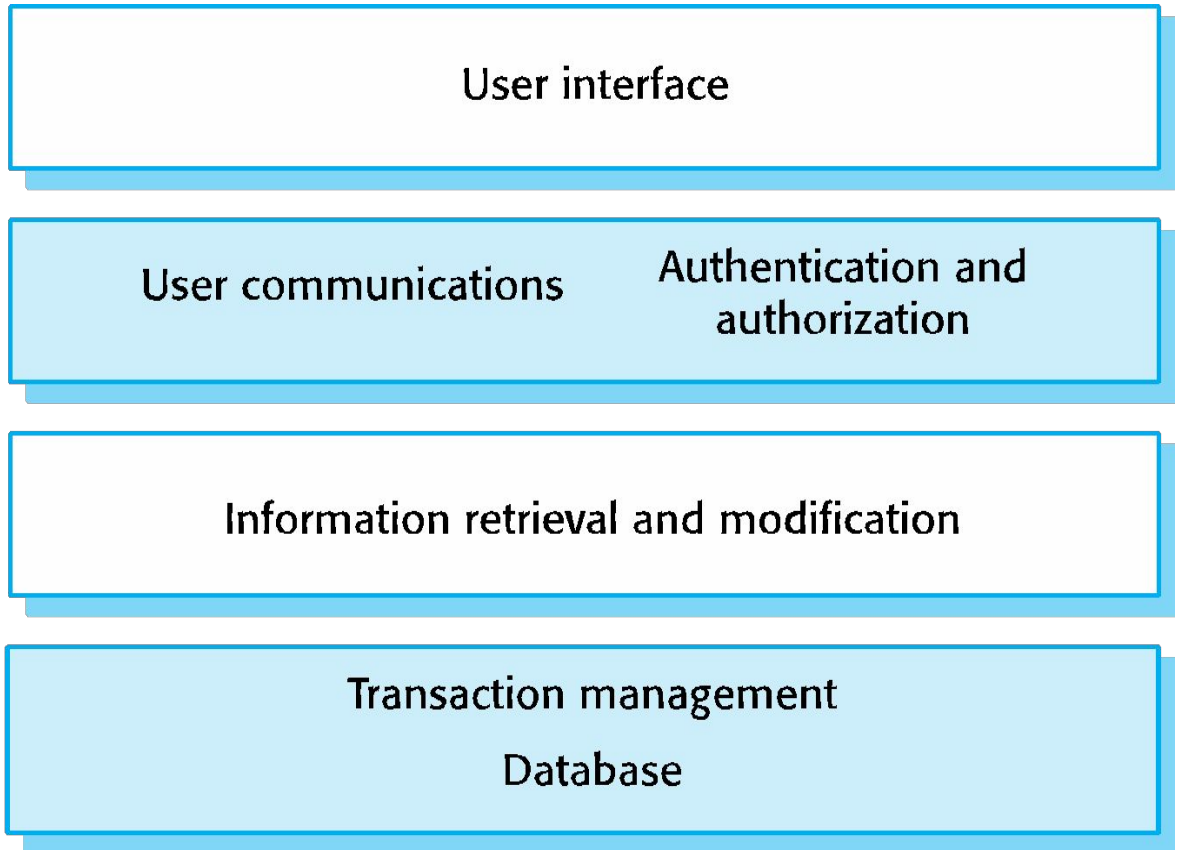
Information systems architecture



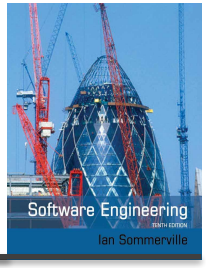
- Information systems have a generic architecture that can be organised as a layered architecture.
- These are transaction-based systems as interaction with these systems generally involves database transactions.
- Layers include:
  - The user interface
  - User communications
  - Information retrieval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

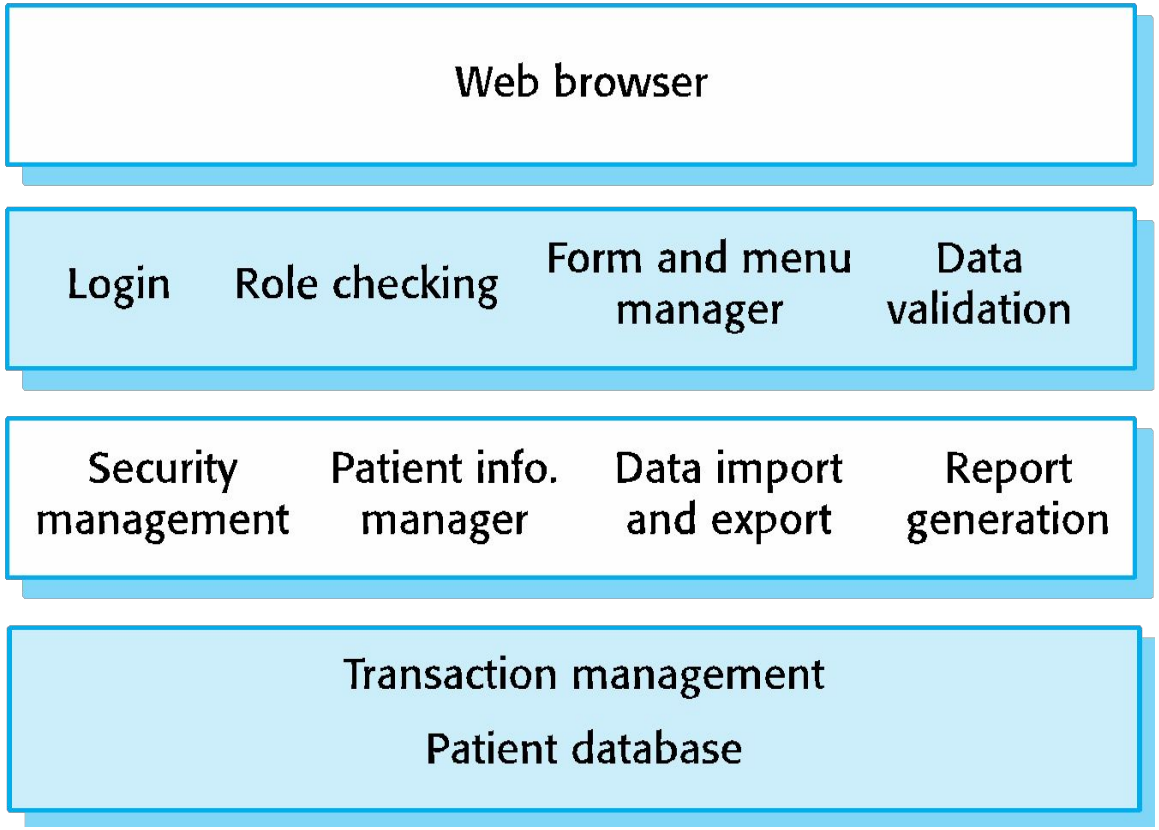
Layered information system architecture



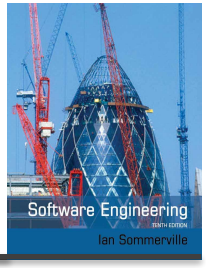


The architecture of the Mentcare system



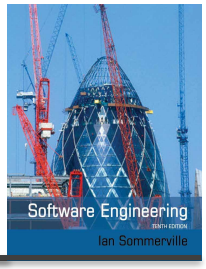


Web-based information systems



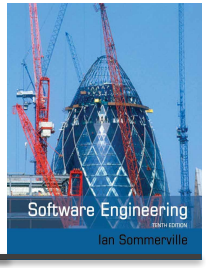
- Information and resource management systems are now usually web-based systems where the user interfaces are implemented using a web browser.
- For example, e-commerce systems are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery 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



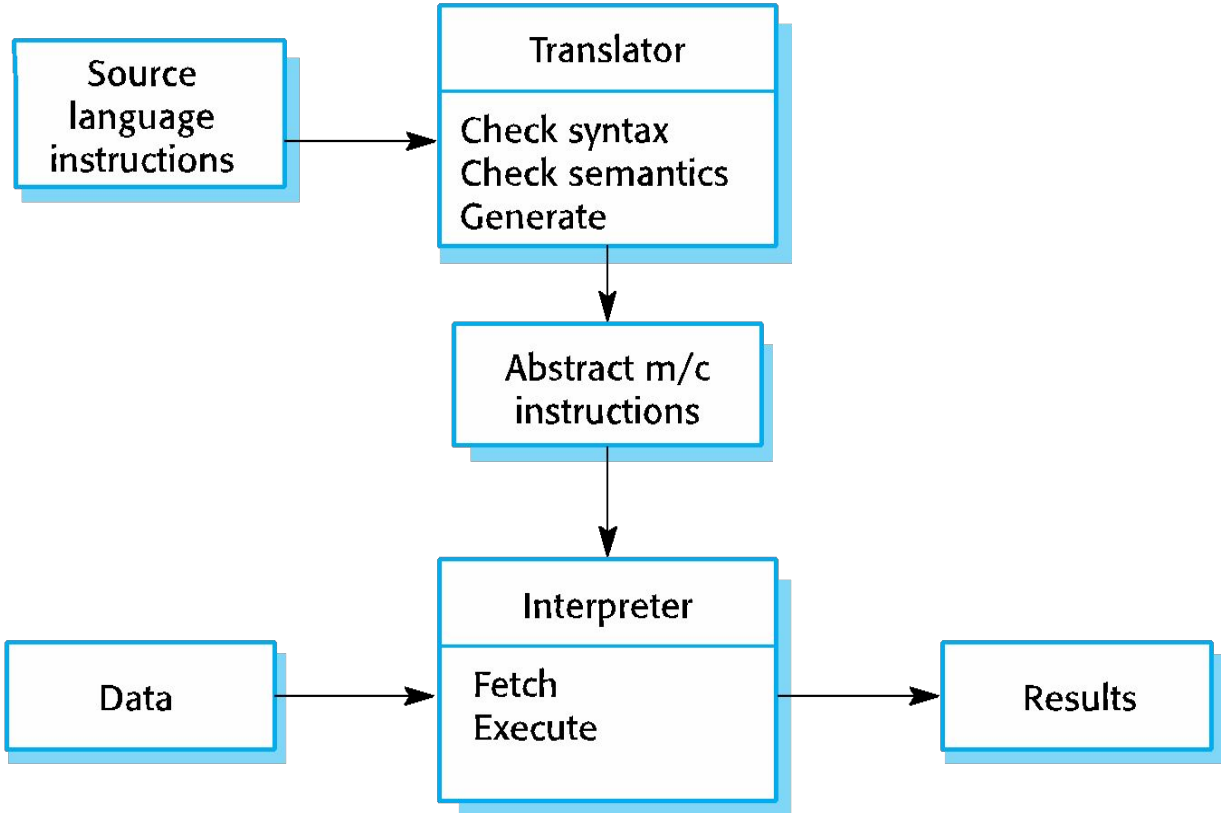
- These systems are often implemented as multi-tier client server/architectures (discussed in Chapter 17)
  - The web server is responsible for all user communications, with the user interface implemented using a web browser;
  - The application server is responsible for implementing application-specific 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

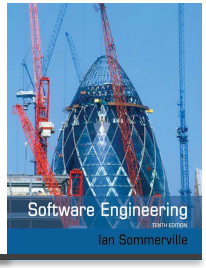


- Accept a natural or artificial language as input and generate some other representation of that language.
- May include an interpreter to act on the instructions in the language that is being processed.
- Used in situations where the easiest way to solve a problem is to describe an algorithm or describe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

The architecture of a language processing system

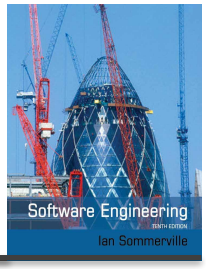


Compiler components



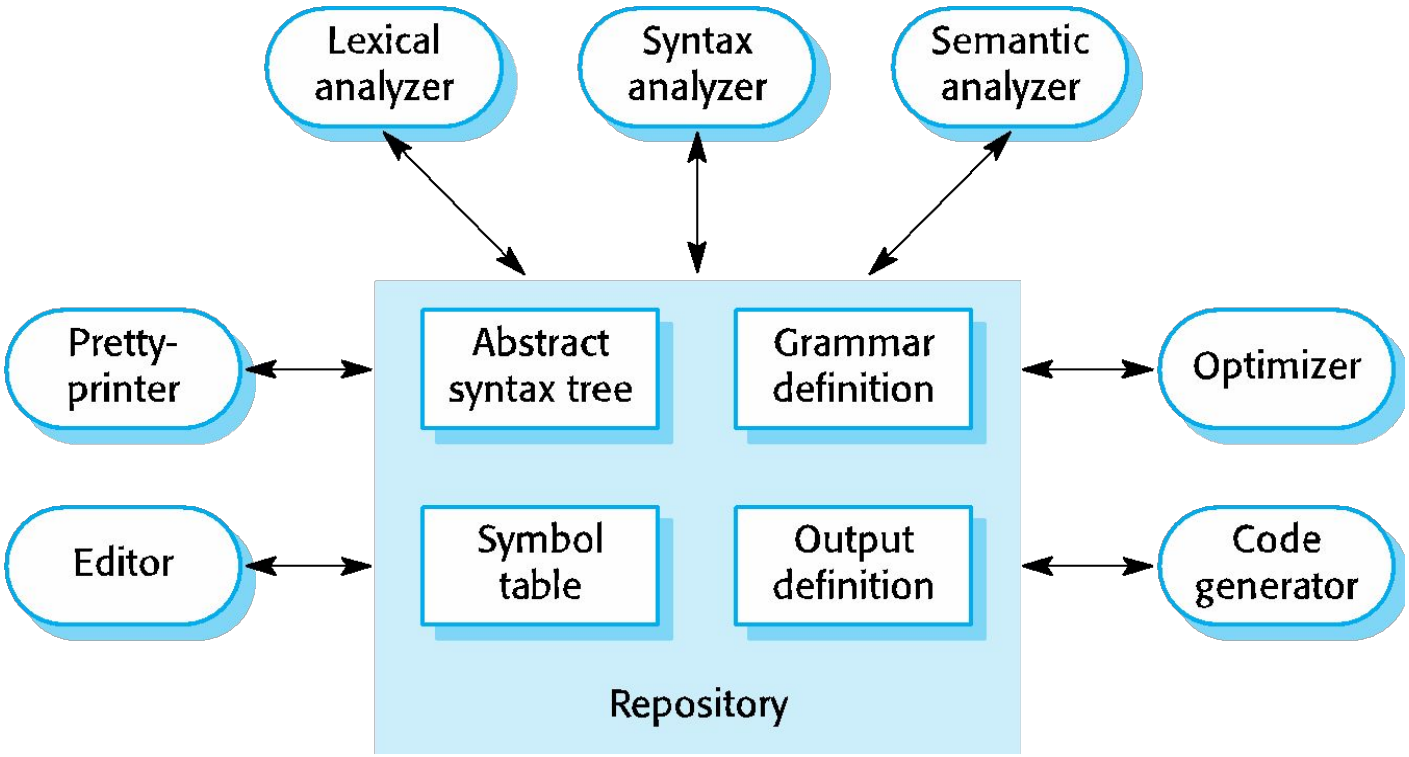
- A lexical analyzer, which takes input language tokens and converts them to an internal form.
- A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
- A syntax analyzer, which checks the syntax of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

Compiler components

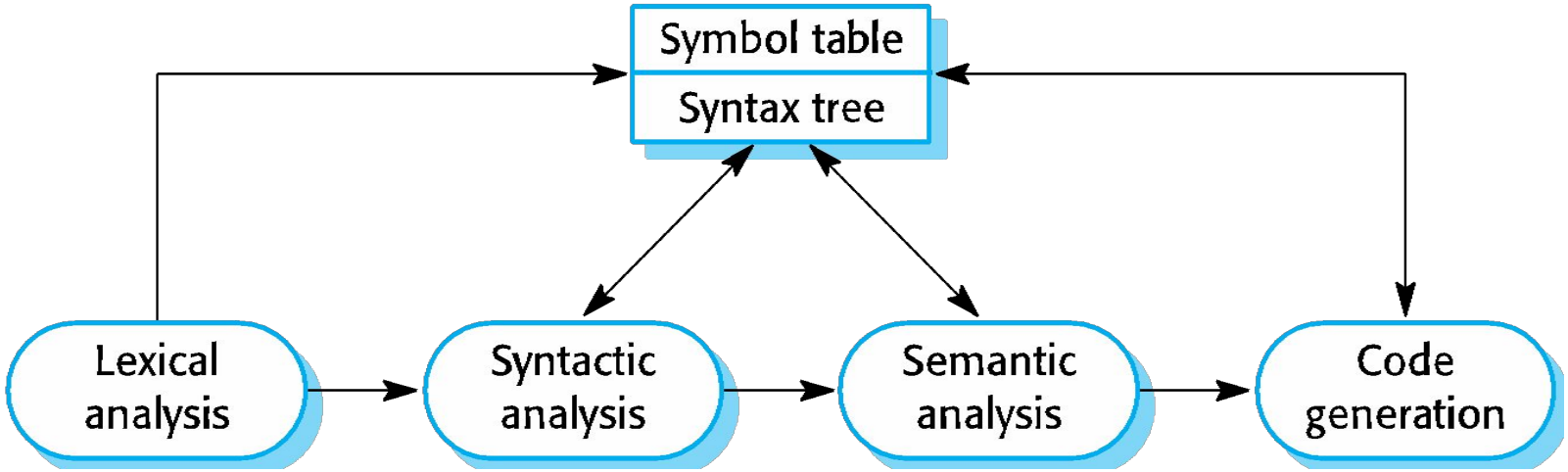


- A semantic analyzer that uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
- A code generator that ‘walks’ the syntax tree and generates abstract machine code.

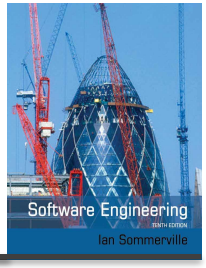
A repository architecture for a language processing system



A pipe and filter compiler architecture

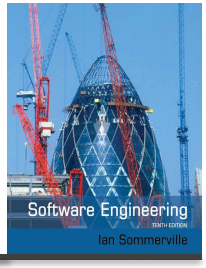


Key points



- 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.

IEEE Standard for Information Technology—Systems Design—Software Design Descriptions

IEEE Computer Society

Sponsored by the
Software & Systems Engineering Standards Committee

1016™

IEEE
3 Park Avenue
New York, NY 10016-5997, USA

20 July 2009

IEEE Std 1016™-2009
(Revision of
IEEE Std 1016-1998)

IEEE Std 1016™-2009
(Revision of
IEEE Std 1016-1998)

IEEE Standard for Information Technology—Systems Design—Software Design Descriptions

Sponsor

Software & Systems Engineering Standards Committee
of the
IEEE Computer Society

Approved 19 March 2009

IEEE-SA Standards Board

Abstract: The required information content and organization for software design descriptions (SDDs) are described. An SDD is a representation of a software design to be used for communicating design information to its stakeholders. The requirements for the design languages (notations and other representational schemes) to be used for conformant SDDs are specified. This standard is applicable to automated databases and design description languages but can be used for paper documents and other means of descriptions.
Keywords: design concern, design subject, design view, design viewpoint, diagram, software design, software design description

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

PDF: ISBN 978-0-7381-5925-6 STD95917
Print: ISBN 978-0-7381-5926-3 STDPD95917

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

Use of an IEEE Standard is wholly voluntary. The IEEE disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEEE Standard document.

The IEEE does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEEE Standards documents are supplied "**AS IS**."

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation, or every ten years for stabilization. When a document is more than five years old and has not been reaffirmed, or more than ten years old and has not been stabilized, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon his or her independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration. A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered the official position of IEEE or any of its committees and shall not be considered to be, nor be relied upon as, a formal interpretation of the IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Recommendations to change the status of a stabilized standard should include a rationale as to why a revision or withdrawal is required. Comments and recommendations on standards, and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854
USA

Introduction

This introduction is not part of IEEE Std 1016-2009, IEEE Standard for Information Technology—Systems Design—Software Design Descriptions.

This standard specifies requirements on the information content and organization for software design descriptions (SDDs). An SDD is a representation of a software design that is to be used for recording design information, addressing various design concerns, and communicating that information to the design's stakeholders.

SDDs play a pivotal role in the development and maintenance of software systems. During its lifetime, an SDD is used by acquirers, project managers, quality assurance staff, configuration managers, software designers, programmers, testers, and maintainers. Each of these stakeholders has unique needs, both in terms of required design information and optimal organization of that information. Hence, a design description contains the design information needed by those stakeholders.

The standard also specifies requirements on the design languages to be used when producing SDDs conforming to these requirements on content and organization.

The standard specifies that an SDD be organized into a number of design views. Each view addresses a specific set of design concerns of the stakeholders. Each design view is prescribed by a design viewpoint. A viewpoint identifies the design concerns to be focused upon within its view and selects the design languages used to record that design view. The standard establishes a common set of viewpoints for design views, as a starting point for the preparation of an SDD, and a generic capability for defining new design viewpoints thereby expanding the expressiveness of an SDD for its stakeholders.

This standard is intended for use in design situations in which an explicit SDD is to be prepared. These situations include traditional software design and construction activities leading to an implementation as well as "reverse engineering" situations where a design description is to be recovered from an existing implementation.

This standard can be applied to commercial, scientific, military, and other types of software. Applicability is not restricted by size, complexity, or criticality of the software. This standard considers both the software and its system context, including the developmental and operational environment. It can be used where software comprises the system or where software is part of a larger system characterized by hardware, software, and human components and their interfaces.

This standard is applicable whether the SDD is captured using paper documents, automated databases, software development tools, or other media. This standard does not explicitly support, nor is it limited to, use with any particular software design methodology or particular design languages, although it establishes minimum requirements on the selection of those design languages.

This standard can be used with IEEE Std 12207™-2008 [B21];[a] it can also be used in other life cycle contexts.

This standard consists of five clauses, as follows:

Clause 1 defines the scope and purpose of the standard.

Clause 2 provides definitions of terms used within the context of the standard.

[a] The numbers in brackets correspond to those of the Bibliography in Annex A.

Clause 3 provides a framework for understanding SDDs in the context of their preparation and use.

Clause 4 describes the required content and organization of an SDD.

Clause 5 defines several design viewpoints for use in producing SDDs.

Annex A provides a bibliography.

Annex B defines how a design language to be used in an SDD is to be described in a uniform manner.

Annex C contains a template for organizing an SDD conforming to the requirements of this standard.

This standard follows the *IEEE Standards Style Manual*.[b] In particular, the word *shall* identifies requirements that must be satisfied in order to claim conformance with this standard. The verb *should* identifies recommendations, and the verb *may* is used to denote that particular courses of action are permissible.

This revision of the standard is modeled after IEEE Std 1471™-2000 [B20], extending the concepts of view, viewpoint, stakeholder, and concern from that standard to support high-level and detailed design and construction for software. The demarcation between architecture, high-level and detailed design varies from system to system and is beyond the scope of this standard.

Notice to users

Laws and regulations

Users of these documents should consult all applicable laws and regulations. Compliance with the provisions of this standard does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

[b] The *IEEE Standards Style Manual* can be found at http://standards.ieee.org/guides/style/index.html.

Updating of IEEE documents

Users of IEEE standards should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect. In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE Standards Association web site at http://ieeexplore.ieee.org/xpl/standards.jsp, or contact the IEEE at the address listed previously.

For more information about the IEEE Standards Association or the IEEE standards development process, visit the IEEE-SA web site at http://standards.ieee.org.

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: http://standards.ieee.org/reading/ieee/updates/errata/index.html. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations can be accessed at the following URL: http://standards.ieee.org/reading/ieee/interp/index.html.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Participants

At the time this standard was submitted to the IEEE-SA Standards Board for approval, the Software Design Descriptions Working Group had the following membership:

Keith R. Middleton, *Chair (acting)*
Vladan V. Jovanovic, *Chair (emeritus)*
Basil A. Sherlund, *Chair (emeritus)*
Rich Hilliard, *Secretary and Technical Editor*

William Bartholomew
Edward Byrne
Bob Cook
Ed Corlett
Philippe Kruchten
Kathy Land
James Moore
Ira Sachs

The following members of the individual balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Butch Anton
Angela Anuszewski
Chris Bagge
Pieter Botman
Juan Carreon
Lawrence Catchpole
Danila Chernetsov
Keith Chow
S. Claassen
Daniel Conte
David Cornejo
Geoffrey Darnton
Terry Dietz
Antonio Doria
Timothy Ehrler
Kameshwar Eranki
Andre Fournier
Eva Freund
David Friscia
Lewis Gray
Michael Grimley
Randall Groves
John Harauz
Mark Henley
Rutger A. Heunks
Ian Hilliard
Werner Hoelzl
Peter Hung
Atsushi Ito
Mark Jaeger
Piotr Karocki
Dwayne Knirk
George Kyle
Susan Land
Dewitt Latimer
David J. Leciston
Daniel Lindberg
Vincent Lipsio
Edward Mccall
Keith R. Middleton
William Milam
James Moore
Rajesh Murthy
Michael S. Newman
William Petit
Ulrich Pohl
Robert Robinson
Randall Safier
James Sanders
Bartien Sayogo
Robert Schaaf
David J. Schultz
Stephen Schwarm
Raymond Senechal
Luca Spotorno
Thomas Starai
Walter Struppler
Gerald Stueve
Marcy Stutzman
K. Subrahmanyam
Richard Thayer
John Thywissen
Thomas Tullia
Vincent Tume
John Walz
P. Wolfgang
Forrest Wright
Janusz Zalewski
Wenhao Zhu

When the IEEE-SA Standards Board approved this standard on 19 March 2009, it had the following membership:

Robert M. Grow, *Chair*
Thomas Prevost, *Vice Chair*
Steve M. Mills, *Past Chair*
Judith Gorman, *Secretary*

John Barr
Karen Bartleson
Victor Berman
Ted Burse
Richard DeBlasio
Andy Drozd
Mark Epstein
Alexander Gelman
Jim Hughes
Rich Hulett
Young Kyun Kim
Joseph L. Koepfinger*
John Kulick
David Law
Ted Olsen
Glenn Parsons
Ron Petersen
Narayanan Ramachandran
Jon Rosdahl
Sam Sciacca

*Member Emeritus

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

Howard Wolfman, *TAB Representative*
Michael Janezic, *NIST Representative*
Satish Aggarwal, *NRC Representative*

Lisa Perry
IEEE Standards Program Manager, Document Development

Malia Zaman
IEEE Standards Program Manager, Technical Program Development

Contents

IEEE Standard for Information Technology—Systems Design—Software Design Descriptions

1. Overview

1.1 Scope

This standard describes software designs and establishes the information content and organization of a software design description (SDD). An SDD is a representation of a software design to be used for recording design information and communicating that design information to key design stakeholders. This standard is intended for use in design situations in which an explicit SDD is to be prepared. These situations include traditional software construction activities, when design leads to code, and "reverse engineering" situations when a design description is recovered from an existing implementation.

This standard can be applied to commercial, scientific, or military software that runs on digital computers. Applicability is not restricted by the size, complexity, or criticality of the software. This standard can be applied to the description of high-level and detailed designs.

This standard does not prescribe specific methodologies for design, configuration management, or quality assurance. This standard does not require the use of any particular design languages, but establishes requirements on the selection of design languages for use in an SDD. This standard can be applied to the preparation of SDDs captured as paper documents, automated databases, software development tools, or other media.

1.2 Purpose

This standard specifies requirements on the information content and organization of SDDs. The standard specifies requirements for the selection of design languages to be used for SDD, and requirements for documenting design viewpoints to be used in organizing an SDD.

1.3 Intended audience

This standard is intended for technical and managerial stakeholders who prepare and use SDDs. It guides a designer in the selection, organization, and presentation of design information. For an organization developing its own SDD practices, the use of this standard can help to ensure that design descriptions are complete, concise, consistent, interchangeable, appropriate for recording design experiences and lessons learned, well organized, and easy to communicate.

1.4 Conformance

An SDD conforms to this standard if it satisfies all of the requirements in Clause 4 and Clause 5 of this standard. Requirements are denoted by the verb *shall*.

2. Definitions

For the purposes of this standard, the following terms and definitions apply. *The Authoritative Dictionary of IEEE Standards Terms* [B13][3] and IEEE Std 12207™-2008 [B21] should be referenced for terms not defined in this clause.

3.1 design attribute: An element of a design view that names a characteristic or property of a design entity, design relationship, or design constraint. *See also:* **design constraint, design entity**, **design relationship**.

3.2 design concern: An area of interest with respect to a software design.

3.3 design constraint: An element of a design view that names and specifies a rule or restriction on a design entity, design attribute, or design relationship. *See also:* **design attribute, design entity**, **design relationship**.

3.4 design element: An item occurring in a design view that may be any of the following: design entity, design relationship, design attribute, or design constraint.

3.5 design entity: An element of a design view that is structurally, functionally, or otherwise distinct from other elements, or plays a different role relative to other design entities. *See also:* **design view**.

3.6 design overlay: A representation of additional, detailed, or derived design information organized with reference to an existing design view.

3.7 design rationale: Information capturing the reasoning of the designer that led to the system as designed, including design options, trade-offs considered, decisions made, and the justifications of those decisions.

[3] The numbers in brackets correspond to those of the Bibliography in Annex A.

3.8 design relationship: Element of a design view that names a connection or correspondence between design entities. *See also:* **design entity**.

3.9 design stakeholder: An individual, organization, or group (or classes thereof playing the same role) having an interest in, or design concerns relative to, the design of some software item. *See also:* **design concern**.

3.10 design subject: A software item or system for which an SDD will be prepared. *Syn:* **software under design**, **system under design**.

3.11 designer: The stakeholder responsible for devising and documenting the software design.

3.12 design view: A representation comprised of one or more design elements to address a set of design concerns from a specified design viewpoint. *See also:* **design concern, design element, design viewpoint.**

3.13 design viewpoint: The specification of the elements and conventions available for constructing and using a design view. *See also:* **design view**.

3.14 diagram (type): A logically coherent fragment of a design view, using selected graphical icons and conventions for visual representation from an associated design language, to be used for representing selected design elements of interest for a system under design from a single viewpoint *See also:* **design subject**.

3. Conceptual model for software design descriptions

This clause establishes a conceptual model for SDDs. The conceptual model includes basic terms and concepts of SDD, the context in which SDDs are prepared and used, the stakeholders who use them, and how they are used.

3.1 Software design in context

A design is a conceptualization of a design subject (the system under design or software under design) that embodies its essential characteristics; demonstrates a means to fulfill its requirements; serves as a basis for analysis and evaluation and can be used to guide its implementation. (See Abran and Moore [B1] for further discussion.)

NOTE 1—This standard does not establish what a design subject may be. A design subject can be any software item to be constructed or which already exists and is to be analyzed. Examples of possible design subjects include, but are not limited to, systems, subsystems, applications, components, libraries, application frameworks, application program interfaces (APIs), and design pattern catalogs. [4]

An SDD is a work product depicting some design subject of interest. An SDD can be produced to capture one or more levels of concern with respect to its design subject. These levels are usually determined by the design methods in use or the life cycle context; they have names such as “architectural design,” “logical design,” or “physical design.”

An SDD can be prepared and used in a variety of design situations. Typically, an SDD is prepared to support the development of a software item to solve a problem, where this problem has been expressed in terms of a set of requirements. The contents of the SDD can then be traced to these requirements. In other

[4] Notes in text, tables, and figures of a standard are given for information only and do not contain requirements needed to implement this standard.

cases, an SDD is prepared to understand an existing system lacking design documentation. In such cases, an SDD is prepared such that information of interest is captured, organized, presented and disseminated to all interested parties. This information of interest can be used for planning, analysis, implementation and evolution of the software system, by identifying and addressing essential design concerns.

NOTE 2—The generic term *software design description* is used in this standard (1) to retain compatibility with the terminology of its predecessor, IEEE Std 1016-1998, and (2) to refer to a range of work products typically defined in use. For example, software design description covers the following information items identified in ISO/IEC 15289:2006 [B25]:[5] database design description (10.14), database detailed design description (10.15), high-level software design description (10.22), interface description (10.27), low-level software design description (10.29), system description (10.71), and system element description (10.72).

A design concern names any area of interest in the design, pertaining to its development, implementation, or operation. Design concerns are expressed by design stakeholders. Frequently, design concerns arise from specific requirements on the software, others arise from contextual constraints. Typical design concerns include functionality, reliability, performance, and maintainability. Typical design stakeholders include users, developers, software designers, system integrators, maintainers, acquirers, and project managers.

NOTE 3—From a system-theoretic standpoint, an SDD captures the information content of the design space with convenient inputs (design diagrams and specifications produced by designers) and outputs (results of transformations produced by software tools). The design space typically contains alternative designs and design rationale in addition to the minimal information of the current version of design. An interesting property of a design description as a system is that its configuration is subject to dynamic evolution and the respective state space, based on its design elements, is not given in advance but created iteratively in a manner of system analysis by synthesis. The final design synthesis is obtained via successive analysis of intermediate designs. Therefore, an SDD can be considered an open, goal-directed system whose end state is a detailed model of the system under design.

An SDD is organized using design views. A design view addresses one or more of the design concerns.

NOTE 4—The use of multiple views to achieve separation of concerns has a long history in software engineering (see Ross, Goodenough, and Irvine [B33] and Ross [B31]), recently in viewpoint-oriented requirements engineering (see Nuseibeh, Kramer, and Finkelstein [B27]), and particularly relevant to this standard, the use of views to rationally present the results of a design process (see Parnas and Clements [B30]) and their use during design (see Gomma [B11]). The particular formulation here is derived from IEEE Std 1471™-2000 [B20].

Each design view is governed by a design viewpoint. Each design viewpoint focuses on a set of the design concerns and introduces a set of descriptive resources called *design elements* that are used to construct and interpret the design view.

Example:

A viewpoint can introduce familiar design elements such as functions, input, and outputs; these elements are used to construct a functional view.

There are four kinds of design elements: design entities, design relationships, design attributes, and design constraints. A design viewpoint determines the types of design elements to be used in any design views it governs. Each design view is expressed as a collection of instances of design entities, design attributes, design relationships among design entities, and design constraints on those elements. The design information needs of stakeholders of the system under design are to be satisfied through use of these elements.

NOTE 5—Although a view need not be a graph, its content is frequently described using diagrams that may be formalized as extensions or specializations of conceptual graphs of design elements (see ISO draft *Conceptual Graphs* [B22]).

[5] Expressions in parentheses in the remainder of this NOTE refer to the subclauses in ISO/IEC 15289:2006 [B25], where these information items are defined.

It is sometimes useful to gather and present information that does not strictly follow the partitioning of information by design viewpoints. A design overlay is a mechanism to organize and present such additional design information.

Design involves the consideration and evaluation of alternatives and trade-offs among alternatives leading to decisions (see Abran and Moore [B1]). Design decisions and the design rationale for decisions are captured both to aid the understanding of current design stakeholders and to support future decision-making (see IEEE Std 12207-2008 [B21]).

The key concepts of SDD are depicted in Figure 1a and Figure 1b.

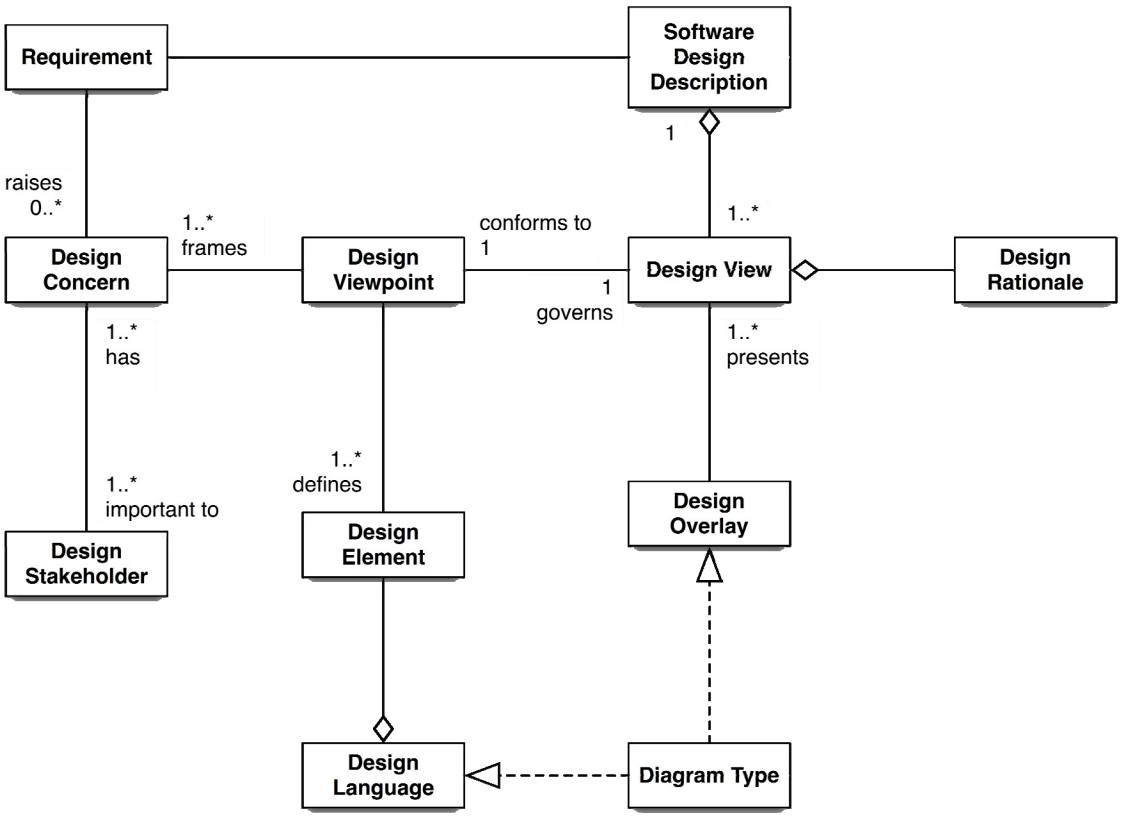


Figure 1a—Conceptual model: top view

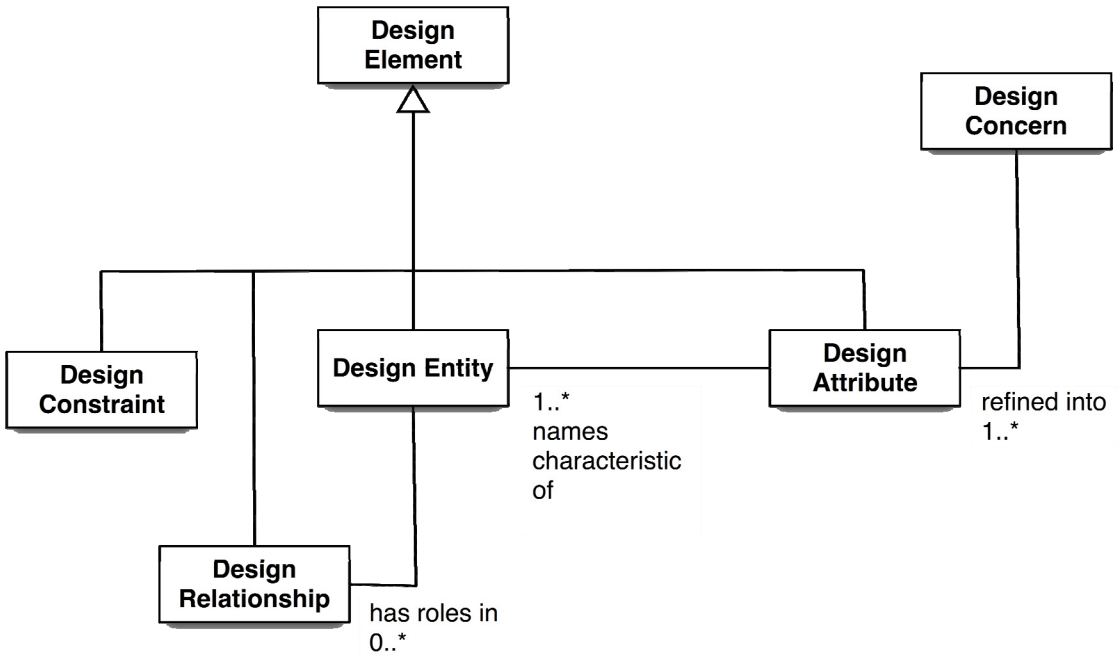


Figure 1b—Conceptual model: design elements

NOTE 6—Figure 1a and Figure 1b provide a summary of the key concepts used in this standard and their relationships. The figure presents these concepts and their relationships from the perspective of a single SDD used to depict a single design subject. An SDD includes one or more design views employing design viewpoints selected to cover stakeholders' design concerns raised by requirements. The contents of each design view is expressed in terms of design entities, their design attributes and design relationships, using selected design languages. In the figure, boxes represent classes of things. Lines connecting boxes represent associations between classes of things. Each class participating in an association has a role in that association. A role is optionally named with a label appearing close to where the association is attached to the class. For example, in the association between Design Viewpoint and Design Element, the role of Design Element is labeled *defines*. Each role may have a multiplicity, denoting a number or set of numbers such as a numeric range. A diamond at the end of an association denotes a part-of relationship. For example, the association between Design View and Software Design Description is interpreted as "one or more Design Views are part of a Software Design Description." This notation is defined in OMG formal/2007-11-2 and 2007-11-4, Unified Modeling Language [B28], [B29].

3.2 Software design descriptions within the life cycle

In this standard, a typical cycle will be used to describe the various design situations in which an SDD can be created and used. This life cycle is based on IEEE Std 12207-2008 [B21].

3.2.1 Influences on SDD preparation

The key software life cycle product that drives a software design is typically the software requirements specification (SRS). An SRS captures the software requirements that will drive the design and design constraints to be considered or observed (see IEEE Std 830™-1998 [B15] and ISO/IEC 15289:2006 [B25]).

3.2.2 influences on software life cycle products

The SDD influences the content of several major software life cycle work products. Developers of these products will be recognized among the SDD's intended audience.

— *Software requirements specification.* Design decisions, or design constraints discovered during the preparation of the SDD, can lead to requirements changes. Maintaining traceability between requirements and design is one way to record the relationship between the two (see Abran and Moore [B1]).

— *Test documentation.* Test plans can be influenced by the SDD. The development of test cases and test procedures can take into consideration the contents of the SDD.

3.2.3 Design verification and design role in validation

Verification is the determination whether a software work product fulfills specified requirements (see IEEE Std 12207-2008 [B21]). An SDD is subject to design verification to ascertain whether the design: addresses stakeholders' design concerns; is consistent with stated requirements; implements intended design decisions (such as those pertaining to interfaces, inputs, outputs, algorithms, resource allocation, and error handling); achieves intended qualities (such as safety, security, or maintainability); and conforms to an imposed architecture. Verification therefore can introduce design concerns to be dealt with in an SDD. Its design verification will usually be performed via review, inspection, or analysis (see IEEE Std 1028™-2008 [B17]).

Validation is the determination that the requirements for a specific intended use of a software work product are fulfilled (see IEEE Std 12207-2008 [B21]). The SDD plays a role in validation by providing: an overview necessary for understanding the implementation; the rationale justifying design decisions made; and traceability back to the requirements on the software under design. (See IEEE Std 1012™-2004 [B16] on verification and validation.)

4. Design description information content

4.1 Introduction

The required contents of an SDD are as follows:

— Identification of the SDD

— Identified design stakeholders

— Identified design concerns

— Selected design viewpoints, each with type definitions of its allowed design elements and design languages

— Design views

— Design overlays

— Design rationale

These are described in the remainder of this clause.

4.2 SDD identification

An SDD shall include the following descriptive information:

- Date of issue and status
- Scope
- Issuing organization
- Authorship (responsibility or copyright information)
- References
- Context
- One or more design languages for each design viewpoint used
- Body
- Summary
- Glossary
- Change history

NOTE—This requirement enables an SDD to meet the guidelines on a description in 7.3 of ISO/IEC 15289:2006 [B25]. The description of design languages is a proper part of the design viewpoint declarations (per 4.5). The body of the SDD is organized into design views (per 4.4), possibly with associated overlays (per 4.7) and design rationale (per 4.8).

4.3 Design stakeholders and their concerns

An SDD shall identify the design stakeholders for the design subject.

An SDD shall identify the design concerns of each identified design stakeholder.

An SDD shall address each identified design concern.

NOTE—An SDD can be used to satisfy the content guidelines for several types of design description as defined in ISO/IEC 15289:2006 [B25], by identifying their content guidelines as design concerns. The types of design descriptions are as follows: database design description (10.14),[6] database detailed design description (10.15), high-level software design description (10.22), interface description (10.27), low-level software design description (10.29), system description (10.71), and system element description (10.72).

4.4 Design views

An SDD shall be organized into one or more design views.

Each design view in an SDD shall conform to its governing design viewpoint.

The purpose of a design view is to address design concerns pertaining to the design subject, to allow a design stakeholder to focus on design details from a specific perspective and effectively address relevant requirements.

[6] Expressions in parentheses in the remainder of this NOTE refer to the subclauses in ISO/IEC 15289:2006 [B25], where the content guidelines for each information item are defined.

Each design view shall address the design concerns specified by its governing design viewpoint.

An SDD is *complete* when each identified design concern is the topic of at least one design view; all design attributes refined from each design concern by some viewpoint have been specified for all of the design entities and relationships in its associated view; and all design constraints have been applied.

An SDD is *consistent* if there are no known conflicts between the design elements of its design views.

NOTE—Users of this standard may wish to state delivery requirements on an SDD in terms of the above notions of completeness and consistency.

4.5 Design viewpoints

For each design view in an SDD, there shall be exactly one design viewpoint governing it.

Each design viewpoint shall be specified by:

- Viewpoint name;
- Design concerns that are the topics of the viewpoint;
- Design elements, defined by that viewpoint, specifically the types of design entities, attributes, relationships, and constraints introduced by that viewpoint or used by that viewpoint (which may have been defined elsewhere). These elements may be realized by one or more design languages;
- Analytical methods or other operations to be used in constructing a design view based upon the viewpoint, and criteria for interpreting and evaluating the design; and
- Viewpoint source (e.g., authorship or citation), when applicable.

In addition, a design viewpoint specification may provide the following information on using the viewpoint:

- Formal or informal consistency and completeness tests to be applied to the view;
- Evaluation or analysis techniques to be applied to a view; and
- Heuristics, patterns, or other guidelines to assist in construction or synthesis of a view.

An SDD shall include a rationale for the selection of each selected viewpoint.

Each design concern identified in an SDD (4.3) shall be framed by at least one design viewpoint selected for use. A design concern may be the focus of more than one viewpoint in an SDD.

NOTE—A design viewpoint specification may be included in the SDD or incorporated by reference.

4.6 Design elements

A design element is any item occurring in a design view. A design element may be any of the following subcases: design entity, design relationship, design attribute, or design constraint.

Each design element in the SDD shall have a name (4.6.2.1), a type (4.6.2.2), and any contents.

NOTE 1—This requirement is “inherited” by the four subcases: design entities, design relationships, design attributes, and design constraints.

The type of each design element shall be introduced within exactly one design viewpoint definition.

A design element may be used in one or more design views.

NOTE 2—A design element is introduced and "owned" by exactly one design view, in accordance with its type definition within the associated viewpoint. It may be shared or referenced within other design views. Sharing of design elements permits the expression of design aspects as in aspect-oriented design.

4.6.1 Design entities

Design entities capture key elements of a software design.

Each design entity shall have a name (4.6.2.1), a type (4.6.2.2), and purpose (4.6.2.3).

Examples of design entities include, but are not limited to, the following: systems, subsystems, libraries, frameworks, abstract collaboration patterns, generic templates, components, classes, data stores, modules, program units, programs, and processes.

NOTE—The number and types of entities needed to express a design view are dependent on a number of factors, such as the complexity of the system, the design technique used, and the tool support environment.

4.6.2 Design attributes

A design attribute names a characteristic or property of a design element (which may be a design entity, design constraint, or a design relationship) and provides a statement of fact about that design element.

All design attributes declared by a design viewpoint shall be specified.

NOTE 1—Design attributes can be thought of as questions about design elements. The answers to those questions are the values of the attributes. All the questions can be answered, but the content of the answer will depend upon the nature of the entity. The collection of answers provides a complete description of an entity. Attribute descriptions should include references and design considerations such as tradeoffs and assumptions when appropriate. In some cases, attribute descriptions may have the value *none*.

NOTE 2—Design attributes have been generalized from the concept of *design entity attribute* (which appeared in IEEE Std 1016-1998 and applied only to design entities) to apply to design entities, design relationships, and design constraints.

NOTE 3—Use of the design attributes in 4.6.2.1 through 4.6.2.3 ensures compatibility with IEEE Std 1016-1998. Other design attributes required as a part of specific design viewpoints are defined with those viewpoints in Clause 5. Some design attributes [such as subordinates (see 5.3.2.2)] can be more usefully represented as design relationships. This was not possible in IEEE Std 1016-1998.

4.6.2.1 Name attribute

The name of the element. Each design element shall have an unambiguous reference name. The names for the elements may be selected to characterize their nature. This will simplify referencing and tracking in addition to providing identification.

4.6.2.2 Type attribute

A description of the kind of element. The type attribute shall describe the nature of the element. It may simply name the kind of element, such as subsystem, component, framework, library, class, subprogram,

module, program unit, function, procedure, process, object, persistent object, class, or data store. Alternatively, design elements may be grouped into major classes to assist in locating an element dealing with a particular type of information.

Within an SDD, the chosen element types shall be applied consistently.

4.6.2.3 Purpose attribute

A description of why the element exists. The purpose attribute shall provide the rationale for the creation of the element.

4.6.2.4 Author attribute

Identification of designer. The author attribute shall identify the designer of the element.

4.6.3 Design relationships

A design relationship names an association or correspondence among two or more design entities. It provides a statement of fact about those design entities.

Each design relationship in an SDD shall have a name (4.6.2.1) and a type (4.6.2.2). A design relationship shall identify the design entities participating in the relationship.

NOTE—There are no design relationships defined in this standard. Most design techniques use design relationships extensively. Normally these design relationships will be defined as a part of a design viewpoint. For example, structured design methods are built around design relationships including input (datum *I* is an **input** to process *A*), output (datum *O* is an **output** of process *A*) and decompose (process *A* **decomposes** into processes *A1*, *A2,* and *A3*) relationships. Object-oriented design methods use design relationships including encapsulation, generalization, specialization, composition, aggregation, realization, and instantiation.

4.6.4 Design constraints

A design constraint is an element of a design view that names a rule or restriction imposed by one design element (the *source*) upon another design element (the *target*), which may be a design entity, design attribute, or design relationship.

Each design constraint in an SDD shall have a name (4.6.2.1) and a type (4.6.2.2). A design constraint shall identify its source and target design entities.

NOTE—There are no design constraints predefined in this standard. Many design techniques introduce design constraints.

4.7 Design overlays

A design overlay is used for presenting additional information with respect to an already-defined design view.

Each design overlay shall be uniquely named and marked as an overlay.

Each design overlay shall be clearly associated with a single viewpoint.

NOTE—Reasons to utilize a design overlay as a part of an SDD include: to provide an extension mechanism for design information to be presented conveniently on top of some view without a requirement for existing external standardization of languages and notations for such representation; to extend expressive power of representation with additional details while reusing information from existing views (i.e., without a need to define additional views or persistently store derivable design information); and to relate design information with facts from the system environment for the convenience of the designer (or other stakeholders).

4.8 Design rationale

Design rationale captures the reasoning of the designer that led to the system as designed and the justifications of those decisions.

Design rationale may take the form of commentary, made throughout the decision process and associated with collections of design elements. Design rationale may include, but is not limited to: design issues raised and addressed in response to design concerns; design options considered; trade-offs evaluated; decisions made; criteria used to guide design decisions; and arguments and justifications made to reach decisions.

NOTE—The only required design rationale is use of the purpose attribute (4.6.2.3).

4.9 Design languages

Design languages are selected as a part of design viewpoint specification (4.5).

A design language may be selected for a design viewpoint only if it supports all design elements defined by that viewpoint.

Design languages shall be selected that have:

- A well-defined syntax and semantics; and
- The status of an available standard or equivalent defining document.

Only standardized and well-established (i.e., previously defined and conveniently available) design languages shall be used in an SDD. In the case of a newly invented design language, the language definition must be provided as a part of the viewpoint declaration.

NOTE 1—Standardized design languages that are in common use are preferable to established languages without a formal definition. Examples of standardized languages include: IDEF0 (IEEE Std 1320.1™-1998 [B18]); IDEF1X (IEEE Std 1320.2™-1998 [B19]); Unified Modeling Language (UML) (OMG [B28] and [B29]); Vienna Definition Method (VDM) (ISO/IEC 13817-1:1996 [B24]); and Z (ISO/IEC 13568:2002 [B23]). Examples of established languages include: state machines, automata, decision tables, Warnier diagrams, Jackson Structured Design (JSD), program design languages (PDL), structure charts, Hierarchy plus Input-Process-Output (HIPO), reliability models, and queuing models.

NOTE 2—It is acceptable to use a design language in more than one view. It is also acceptable to use more than one design language within any number of views when each design language to be used is declared by the viewpoint. This is acceptable even for a portion of the design; for example, when used as a basis for interchange; due to organizational considerations such as development by non-collocated team members; subcontracting of partial design responsibility; or taking advantage of particular design tools or designer expertise.

NOTE 3—Annex B establishes a uniform format for describing design languages to be used in SDDs.

NOTE 4—In case that no adequate design language is readily available for a specified viewpoint, it is the designer's responsibility to provide an adaptation of an existing language or the definition of an appropriate new design language. This design language definition would be provided by the designer to be included in the SDD in accordance with the requirements for viewpoints in 4.5.

5. Design viewpoints

5.1 Introduction

This clause defines several design viewpoints for use in SDDs. It illustrates the realization of these design viewpoints in terms of design language selections, relates design concerns with viewpoints, and establishes language (notation and method) neutral names for these viewpoints. Table 1 summarizes these design viewpoints. For each viewpoint, its name, design concerns, and appropriate design languages are listed. Short descriptions relating a minimal set of design entities, design relationships, design entity attributes, and design constraints are provided for each viewpoint. Additional references pertinent to the use of each viewpoint are also listed.

A design viewpoint defined in this clause shall be used in the SDD whenever applicable to the design subject, based on identified design concerns (4.3).

Table 1—Summary of design viewpoints

Design viewpoint	Design concerns	Example design languages
Context (5.2)	Systems services and users	IDEF0, UML use case diagram, Structured Analysis context diagram
Composition (5.3) Can be refined into new viewpoints, such as: functional (logical) decomposition, and run-time (physical) decomposition.	Composition and modular assembly of systems in terms of subsystems and (pluggable) components, buy vs. build, reuse of components	Logical: UML package diagram, UML component diagram, Architecture Description Languages, IDEF0, Structure chart, HIPO Physical: UML deployment diagram
Logical (5.4)	Static structure (classes, interfaces, and their relationships) Reuse of types and implementations (classes, data types)	UML class diagram, UML object diagram
Dependency (5.5)	Interconnection, sharing, and parameterization	UML package diagram and component diagram
Information (5.6) with data distribution overlay and physical volumetric overlay	Persistent information	IDEF1X, entity-relation diagram, UML class diagram
Patterns (5.7)	Reuse of patterns and available Framework template	UML composite structure diagram
Interface (5.8)	Service definition, service access	Interface definition languages (IDL), UML component diagram
Structure (5.9)	Internal constituents and organization of design subjects, components and classes	UML structure diagram, class diagram
Interaction (5.10)	Object communication, messaging	UML sequence diagram, UML communication diagram
State dynamics (5.11)	Dynamic state transformation	UML state machine diagram, statechart (Harel's), state transition table (matrix), automata, Petri net
Algorithm (5.12)	Procedural logic	Decision table, Warnier diagram, JSP, PDL
Resources (5.13) May be refined into resource based viewpoints with possible overlays	Resource utilization	UML Real-time Profile, UML class diagram, UML Object Constraint Language (OCL)

5.2 Context viewpoint

The Context viewpoint depicts services provided by a design subject with reference to an explicit context. That context is defined by reference to actors that include users and other stakeholders, which interact with the design subject in its environment. The Context viewpoint provides a "black box" perspective on the design subject.

Services depict an inherently functional aspect or anticipated cases of use of the design subject (hence "use cases" in UML). Stratification of services and their descriptions in the form of scenarios of actors' interactions with the system provide a mechanism for adding detail. Services may also be associated with actors through information flows. The content and manner of information exchange with the environment implies additional design information and the need for additional viewpoints (see 5.10).

A Deployment overlay to a Context view can be transformed into a Deployment view whenever the execution hardware platform is part of the design subject; for stand-alone software design, a Deployment overlay maps software entities onto externally available entities not subject of the current design effort. Similarly, work allocation to teams and other management perspectives are overlays in the design.

5.2.1 Design concerns

The purpose of the Context viewpoint is to identify a design subject's offered services, its actors (users and other interacting stakeholders), to establish the system boundary and to effectively delineate the design subject's scope of use and operation.

Drawing a boundary separating a design subject from its environment, determining a set of services to be provided, and the information flows between design subject and its environment, is typically a key design decision. That makes this viewpoint applicable to most design efforts.

When the system is portrayed as a black box, with internal decisions hidden, the Context view is often a starting point of design, showing what is to be designed functionally as the only available information about the design subject: a name and an associated set of externally identifiable services. Requirements analysis identifies these services with the specification of quality of service attributes, henceforth invoking many non-functional requirements. Frequently incomplete, a Context view is begun in requirements analysis. Work to complete this view continues during design.

5.2.2 Design elements

Design entities: actors—external active elements interacting with the design subject, including users, other stakeholders and external systems, or other items; services—also called *use cases;* and directed information flows between the design subject, treated as a black box, and its actors associating actors with services. Flows capture the expected information content exchanged.

Design relationships: receive output and provide input (between actors and the design subject).

Design constraints: qualities of service; form and medium of interaction (provided to and received from) with environment.

5.2.3 Example languages

Any black-box type diagrams can be used to realize the Context viewpoint. Appropriate languages include Structured Analysis [e.g., IDEF0 (IEEE Std 1320.1-1998 [B18]), Structured Analysis and Design

Technique (SADT) (Ross [B32]) or those of the DeMarco or Gane-Sarson variety], the Cleanroom's black box diagrams, and UML use cases (OMG [B28]).

5.3 Composition viewpoint

The Composition viewpoint describes the way the design subject is (recursively) structured into constituent parts and establishes the roles of those parts.

5.3.1 Design concerns

Software developers and maintainers use this viewpoint to identify the major design constituents of the design subject, to localize and allocate functionality, responsibilities, or other design roles to these constituents. In maintenance, it can be used to conduct impact analysis and localize the efforts of making changes. Reuse, on the level of existing subsystems and large-grained components, can be addressed as well. The information in a Composition view can be used by acquisition management and in project management for specification and assignment of work packages, and for planning, monitoring, and control of a software project. This information, together with other project information, can be used in estimating cost, staffing, and schedule for the development effort. Configuration management may use the information to establish the organization, tracking, and change management of emerging work products (see IEEE Std 12207-2008 [B21]).

5.3.2 Design elements

Design entities: types of constituents of a system: subsystems, components, modules; ports and (provided and required) interfaces; also libraries, frameworks, software repositories, catalogs, and templates.

Design relationships: composition, use, and generalization. The Composition viewpoint supports the recording of the part-whole relationships between design entities using realization, dependency, aggregation, composition, and generalization relationships. Additional design relationships are required and provided (interfaces), and the attachment of ports to components.

Design attributes: For each design entity, the viewpoint provides a reference to a detailed description via the identification attribute. The attribute descriptions for identification, type, purpose, function, and definition attribute should be utilized.

5.3.2.1 Function attribute

A statement of what the entity does. The function attribute states the transformation applied by the entity to its inputs to produce the output. In the case of a data entity, this attribute states the type of information stored or transmitted by the entity.

NOTE—This design attribute is retained for compatibility with IEEE Std 1016-1998.

5.3.2.2 Subordinates attribute

The identification of all entities composing this entity. The subordinates attribute identifies the "composed of" relationship for an entity. This information is used to trace requirements to design entities and to identify parent/child structural relationships through a design subject.

NOTE—This design attribute is retained for compatibility with IEEE Std 1016-1998. An equivalent capability is available through the composition relationship.

5.3.3 Example languages

UML component diagrams (see OMG [B28]) cover this viewpoint. The simplest graphical technique used to describe functional system decomposition is a hierarchical decomposition diagram; such diagram can be used together with natural language descriptions of purpose and function for each entity, such as is provided by IDEF0 (IEEE Std 1320.1-1998 [B18]), the Structure Chart (Yourdon and Constantine [B38], and the HIPO Diagram. Run-time composition can also use structured diagrams (Page-Jones [B29]).

5.4 Logical viewpoint

The purpose of the Logical viewpoint is to elaborate existing and designed types and their implementations as classes and interfaces with their structural static relationships. This viewpoint also uses examples of instances of types in outlining design ideas.

5.4.1 Design concerns

The Logical viewpoint is used to address the development and reuse of adequate abstractions and their implementations. For any implementation platform, a set of types is readily available for the domain abstractions of interest in a design subject, and a number of new types is to be designed, some of which may be considered for reuse. The main concern is the proper choice of abstractions and their expression in terms of existing types (some of which may had been specific to the design subject).

5.4.2 Design elements

Design entities: class, interface, power type, data type, object, attribute, method, association class, template, and namespace.

Design relationships: association, generalization, dependency, realization, implementation, instance of, composition, and aggregation.

Design attributes: name, role name, visibility, cardinality, type, stereotype, redefinition, tagged value, parameter, and navigation efficiency.

Design constraints: value constraints, relationships exclusivity constraints, navigability, generalization sets, multiplicity, derivation, changeability, initial value, qualifier, ordering, static, pre-condition, post-condition, and generalization set constraints.

5.4.3 Example languages

UML class diagrams and UML object diagrams (showing objects as instances of their respective classes) (OMG [B28]). Lattices of types and references to implemented types are commonly used as supplementary information.

5.5 Dependency viewpoint

The Dependency viewpoint specifies the relationships of interconnection and access among entities. These relationships include shared information, order of execution, or parameterization of interfaces.

5.5.1 Design concerns

A Dependency view provides an overall picture of the design subject in order to assess the impact of requirements or design changes. It can help maintainers to isolate entities causing system failures or resource bottlenecks. It can aid in producing the system integration plan by identifying the entities that are needed by other entities and that must be developed first. This description can also be used by integration testing to aid in the production of integration test cases.

5.5.2 Design elements

Design entities: subsystem, component, and module.

Design relationships: uses, provides, and requires.

Design attribute: name (4.6.2.1), type (4.6.2.2), purpose (4.6.2.3), dependencies (5.5.2.1), and resources. These attributes should be provided for all design entities.

5.5.2.1 Dependencies attribute

A description of the relationships of this entity with other entities. The dependencies attribute identifies the uses or requires the presence of relationship for an entity. This attribute is used to describe the nature of each interaction including such characteristics as timing and conditions for interaction. The interactions involve the initiation, order of execution, data sharing, creation, duplicating, usage, storage, or destruction of entities.

NOTE—This design entity attribute is retained for compatibility with IEEE Std 1016-1998.

5.5.3 Example languages

UML component diagrams and UML package diagrams showing dependencies among subsystems (OMG [B28]).

5.6 Information viewpoint

The Information viewpoint is applicable when there is a substantial persistent data content expected with the design subject.

5.6.1 Design concerns

Key concerns include persistent data structure, data content, data management strategies, data access schemes, and definition of metadata.

5.6.2 Design elements

Design entities: data items, data types and classes, data stores, and access mechanisms.

Design relationships: association, uses, implements. Data attributes, their constraints and static relationships among data entities, aggregates of attributes, and relationships.

Design attributes: persistence and quality properties.

5.6.2.1 Data attribute

A description of data elements internal to the entity. The data attribute describes the method of representation, initial values, use, semantics, format, and acceptable values of internal data. The description of data may be in the form of a data dictionary that describes the content, structure, and use of all data elements. Data information should describe everything pertaining to the use of data or internal data structures by this entity. It should include data specifications such as formats, number of elements, and initial values. It should also include the structures to be used for representing data such as file structures, arrays, stacks, queues, and memory partitions.

The meaning and use of data elements should be specified. This description includes such things as static versus dynamic, whether it is to be shared by transactions, used as a control parameter, or used as a value, loop iteration count, pointer, or link field. In addition, data information should include a description of data validation needed for the process.

NOTE—This design attribute is retained for compatibility with IEEE Std 1016-1998.

5.6.3 Example languages

IDEF1X (IEEE Std 1320.2™-1998 [B19]), UML class diagrams (OMG [B28]).

5.7 Patterns use viewpoint

This viewpoint addresses design ideas (emergent concepts) as collaboration patterns involving abstracted roles and connectors.

5.7.1 Design concerns

Key concerns include reuse at the level of design ideas (design patterns), architectural styles, and framework templates.

5.7.2 Design elements

Design entities: collaboration, class, connector, role, framework template, and pattern.

Design relationships: association, collaboration use, and connector.

Design attributes: name.

Design constraints: collaboration constraints.

5.7.3 Example languages

UML composite structure diagram and a combination of the UML class diagram and the UML package diagram (OMG [B28]).

5.8 Interface viewpoint

The Interface viewpoint provides information designers, programmers, and testers the means to know how to correctly use the services provided by a design subject. This description includes the details of external and internal interfaces not provided in the SRS. This viewpoint consists of a set of interface specifications for each entity.

NOTE—User interfaces are addressed separately.

5.8.1 Design concerns

An Interface view description serves as a binding contract among designers, programmers, customers, and testers. It provides them with an agreement needed before proceeding with the detailed design of entities. The interface description is used by technical writers to produce customer documentation or may be used directly by customers. In the latter case, the interface description could result in the production of a human interface view.

Designers, programmers, and testers often use design entities that they did not develop. These entities can be reused from earlier projects, contracted from an external source, or produced by other developers. The interface description establishes an agreement among designers, programmers, and testers about how cooperating entities will interact. Each entity interface description should contain everything another designer or programmer needs to know to develop software that interacts with that entity. A clear description of entity interfaces is essential on a multi-person development for smooth integration and ease of maintenance.

5.8.2 Design elements

The attributes for identification (4.6.2.1), function (5.3.2.1), and interface (5.8.2.1) should be provided for all design entities.

5.8.2.1 Interface attribute

A description of how other entities interact with this entity. The interface attribute describes the methods of interaction and the rules governing those interactions. Methods of interaction include the mechanisms for invoking or interrupting the entity, for communicating through parameters, common data areas or messages, and for direct access to internal data. The rules governing the interaction include the communications protocol, data format, acceptable values, and the meaning of each value.

This attribute provides a description of the input ranges, the meaning of inputs and outputs, the type and format of each input or output, and output error codes. For information systems, it should include inputs, screen formats, and a complete description of the interactive language.

NOTE—This design attribute is retained for compatibility with IEEE Std 1016-1998

5.8.3 Example languages

Interface definition languages (IDL), UML component diagram (OMG [B28]). In case of user interfaces the Interface view should include screen formats, valid inputs, and resulting outputs. For data-driven entities, a data dictionary should be used to describe the data characteristics. Those entities that are highly visible to a user and involve the details of how the customer should perceive the system should include a functional model, scenarios for use, detailed feature sets, and the interaction language.

5.9 Structure viewpoint

The Structure viewpoint is used to document the internal constituents and organization of the design subject in terms of like elements (recursively).

5.9.1 Design concerns

Compositional structure of coarse-grained components and reuse of fine-grained components.

5.9.2 Design elements

Design entities: port, connector, interface, part, and class.

Design relationships: connected, part of, enclosed, provided, and required.

Design attributes: name, type, purpose, and definition.

Design constraints: interface constraints, reusability constraints, and dependency constraints.

5.9.3 Example languages

UML composite structure diagram, UML class diagram, and UML package diagram (OMG [B28]).

5.10 Interaction viewpoint

The Interaction viewpoint defines strategies for interaction among entities, regarding why, where, how, and at what level actions occur.

5.10.1 Design concerns

For designers. this includes evaluating allocation of responsibilities in collaborations, especially when adapting and applying design patterns; discovery or description of interactions in terms of messages among affected objects in fulfilling required actions; and state transition logic and concurrency for reactive, interactive, distributed, real-time, and similar systems.

5.10.2 Design elements

Classes, methods. states, events, signals, hierarchy, concurrency, timing, and synchronization.

5.10.3 Examples

UML composite structure diagram, UML interaction diagram (OMG [B28]).

5.11 State dynamics viewpoint

Reactive systems and systems whose internal behavior is of interest use this viewpoint.

5.11.1 Design concerns

System dynamics including modes, states, transitions, and reactions to events.

5.11.2 Design elements

Design entities: event, condition, state, transition, activity, composite state, submachine state, critical region, and trigger.

Design relationships: part-of, internal, effect, entry, exit, and attached to.

Design attributes: name, completion, active, initial, and final.

Design constraints: guard conditions, concurrency, synchronization, state invariant, transition constraint, and protocol.

5.11.3 Example languages

UML state machine diagram (OMG [B28]), Harel statechart, state transition table (matrix), automata, Petri net.

5.12 Algorithm viewpoint

The detailed design description of operations (such as methods and functions), the internal details and logic of each design entity.

5.12.1 Design concerns

The Algorithm viewpoint provides details needed by programmers, analysts of algorithms in regard to time-space performance and processing logic prior to implementation, and to aid in producing unit test plans.

5.12.2 Design elements

These should include the attribute descriptions for identification, processing (5.12.1), and data for all design entities.

5.12.3 Processing attribute

A description of the rules used by the entity to achieve its function. The processing attribute describes the algorithm used by the entity to perform a specific task and its contingencies. This description is a refinement of the function attribute and is the most detailed level of refinement for the entity.

This description should include timing, sequencing of events or processes, prerequisites for process initiation, priority of events, processing level, actual process steps, path conditions, and loop back or loop termination criteria. The handling of contingencies should describe the action to be taken in the case of overflow conditions or in the case of a validation check failure.

NOTE—This design attribute is retained for compatibility with IEEE Std 1016-1998.

5.12.4 Examples

Decision tables and flowcharts; program design languages, "pseudo-code," and (actual) programming languages may also be used.

5.13 Resource viewpoint

The purpose of the Resource viewpoint is to model the characteristics and utilization of resources in a design subject.

5.13.1 Design concerns

Key concerns include resource utilization, resource contention, availability, and performance.

5.13.2 Design elements

Design entities: resources, usage policies.

Design relationships: allocation and uses.

Design attributes: identification (4.6.2.1), resource (5.13.2.1), performance measures (such as throughput, rate of consumption).

Design constraints: priorities, locks, resource constraints.

5.13.2.1 Resources attribute

A description of the elements used by the entity that are external to the design. The resources attribute identifies and describes all of the resources external to the design that are needed by this entity to perform its function. The interaction rules and methods for using the resource are to be specified by this attribute.

This attribute provides information about items such as physical devices (printers, disc-partitions, memory banks), software services (math libraries, operating system services), and processing resources (CPU cycles, memory allocation, buffers).

The resources attribute should describe usage characteristics such as the processing time at which resources are to be acquired and sizing to include quantity, and physical sizes of buffer usage. It should also include the identification of potential race and deadlock conditions as well as resource management facilities.

NOTE—This design attribute is retained for compatibility with IEEE Std 1016-1998.

5.13.3 Examples

Woodside [B37], UML class diagram, UML Object Constraint Language (OMG [B28]).

Annex A

(informative)

Bibliography

[B1] Abran, A., and J. W. Moore (editors), *Guide to the Software Engineering Body of Knowledge,* 2004 Edition. IEEE Press, 2005.

[B2] Arlow, J., and I. Neustadt, *Enterprise Patterns and MDA: Building better software with archetype patterns and UML.* Addison-Wesley, 2004.

[B3] Coplien, J., *Multi-Paradigm Design for C++.* Addison-Wesley, 1998.

[B4] D'Souza, D., and A. Wills, *Objects, Components, and Frameworks with UML.* Addison-Wesley 1999.

[B5] Douglass, B. P., *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns.* Addison-Wesley, 1999.

[B6] Douglass, B. P., *Real-Time UML.* Third edition, Addison-Wesley, 2004.

[B7] Evitts, P., A *UML Pattern Language.* Sams, 2000.

[B8] Fayad, M. E., and R. E. Johnson (editors), *Domain-Specific Application Frameworks.* Wiley, 2000.

[B9] Feldmann, C. G., *The Practical Guide to Business Process Reengineering Using IDEF0.* Dorset House Publishing, 1998.

[B10] Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[B11] Gomma, H., *Designing Software Product Lines with UML.* Addison-Wesley, 2005.

[B12] Graham, I., *Object-Oriented Methods.* Addison-Wesley, 2001.

[B13] IEEE 100™, *The Authoritative Dictionary of IEEE Standards Terms,* Seventh Edition. New York: Institute of Electrical and Electronics Engineers, Inc.[7, 8]

[B14] IEEE Std 610.12™-1990, IEEE Standard Glossary of Software Engineering Terminology.

[B15] IEEE Std 830™-1998, IEEE Recommended Practice for Software Requirements Specifications.

[B16] IEEE Std 1012™-2004, IEEE Standard for Software Verification and Validation.

[B17] IEEE Std 1028™-2008, IEEE Standard for Software Reviews and Audits.

[B18] IEEE Std 1320.1™-1998, IEEE Standard for Functional Modeling Language—Syntax and Semantics for IDEF0.

[B19] IEEE Std 1320.2™-1998, IEEE Standard Conceptual Modeling Language—Syntax and Semantics for IDEF1X97 (IDEFobject).

[B20] IEEE Std 1471™-2000, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems.[9]

[B21] IEEE Std 12207™-2008 Systems and software engineering—Software life cycle processes.[10]

[7] IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).
[8] The IEEE standards or products referred to in Annex A are trademarks owned by the Institute of Electrical and Electronics Engineers, Incorporated.
[9] IEEE Std 1471-2000 is also known as ISO/IEC 42010:2007.
[10] IEEE Std 12207-2008 is also known as ISO/IEC 12207:2008.

[B22] ISO draft standard, Conceptual Graphs, 2001.[11]

[B23] ISO/IEC 13568:2002, Information technology—Z formal specification notation—Syntax, type system and semantics.[12]

[B24] ISO/IEC 13817–1:1996, Information technology—Programming languages, their environments and system software interfaces—Vienna Development Method—Specification Language—Part 1: Base language.

[B25] ISO/IEC 15289:2006, Systems and software engineering—Content of systems and software life cycle process information products (Documentation).

[B26] Kruchten, P., *The Rational Unified Process.* Addison-Wesley, 2000.

[B27] Nuseibeh, B., J. Kramer, and A. Finkelstein, "A framework for expressing the relationships between multiple views in requirements specification," *IEEE Transactions on Software Engineering,* 20(10) pp. 760–773, 1994.

[B28] OMG formal/2007-11-02, Unified Modeling Language (OMG UML), Superstructure, version 2.1.2, November 2007.

[B29] OMG formal/2007-11-04, Unified Modeling Language (OMG UML), Infrastructure, version 2.1.2, November 2007.

[B30] Page-Jones, M., *The Practical Guide to Structured Systems Design,* Second Edition. Prentice Hall, 1988.

[B31] Parnas, D. L., and P. C. Clements, "A rational design process: how and why to fake it," *IEEE Transactions on Software Engineering,* 12(7), 1986.

[B32] Ross, D. T., "Structured Analysis: a language for communicating ideas," *IEEE Transactions on Software Engineering,* 1977.

[B33] Ross, D. T., J. B. Goodenough, and C.A. Irvine, "Software engineering: Process, principles, and goals," *COMPUTER* 8(5) (May 1975): 17–27

[B34] Rumbaugh, J., I. Jacobson, and G. Booch, *UML Reference Manual,* Second Edition, Addison-Wesley, 2005.

[B35] Shon, D., *The Reflective Practitioner.* Basic Books, 1983.

[B36] Weiss, D., and C. Lai, *Software Product Line Engineering.* Addison-Wesley, 1999.

[B37] Woodside, C. M., "A three-view model for performance engineering of concurrent software," *IEEE Transactions on Software Engineering,* 21(9), 1995.

[B38] Yourdon, E., and L. Constantine, *Structured Design.* Prentice Hall, 1979.

[11] Available at http://www.jfsowa.com/cg/cgstand.htm.

[12] ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch/). ISO/IEC publications are also available in the United States from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112, USA (http://global.ihs.com/). Electronic copies are available in the United States from the American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

Annex B

(informative)

Conforming design language description

This annex defines a uniform format for describing design languages. Any design language can be documented in terms of a number of characteristics. These characteristics have been chosen to facilitate the selection of design languages in the selection and definition of viewpoints (Clause 4). The format has a structured textual format intended to allow both human and machine-readable application.

It is envisioned that the providers of design languages (whether commercial, industrial, research or experimental) will be able to document the intended usage of the design language using this format. This documentation will allow designers to more readily review the properties of that design language for use in an SDD because the attributes captured here match the considerations to be made when defining a design viewpoint in accordance with Clause 4 and Clause 5 of this standard.

Every well-formed design language description should specify the design language name.

Examples of design language names:

"IDEF0" "UML state machine"

Every well-formed design language description must have a reference to the definition of the design language. This may be a reference to a standard or other defining document.

Example of design language reference:

IEEE Std 1302.1, OMG-Unified Modeling Language, v1.4 September 2001

Every well-formed design language description should contain an identification of one or more design concerns that are capable of being expressed using this design language. This information can be used by designers to choose appropriate design languages to implement selected design viewpoints within an SDD.

Examples of design concerns:

Functionality, Reliability (for additional examples see Clause 5).

Every well-formed design language description must identify each design entity type defined by the design language.

Examples of design elements:

State, Transition, Event (for additional examples see Clause 5).

Every well-formed design language description must identify each design entity attribute type, and the design entity that define it.

Examples of design attributes:

transitionLabel *defined by*: Transition; guardCondition *defined by*: Transition (for additional examples see Clause 4 and Clause 5).

Every well-formed design language description must identify the design entity relationship types that are a part of the design language. First, the relationship type is named. Then the design entity types participating in the relationship are identified.

Examples of design relationships:

subActivities participants: 2 or more Activities (for additional examples see Clause 5).

Annex C

(informative)

Templates for an SDD

The template in Figure C.1 shows one possible way to organize and format an SDD conforming to the requirements of Clause 4.

- Frontspiece
  - Date of issue and status
  - Issuing organization
  - Authorship
  - Change history
- Introduction
  - Purpose
  - Scope
  - Context
  - Summary
- References
- Glossary
- Body
  - Identified stakeholders and design concerns
  - Design viewpoint 1
  - Design view 1
  - …
  - Design viewpoint n
  - Design view n
  - Design rationale

Figure C.1—Table of contents for an SDD